PROYECTO INTEGRADOR DE LA CARRERA DE INGENIERÍA MECÁNICA

IMPLEMENTACIÓN DE MODELOS LATTICE BOLTZMANN PARA FLUJO MULTIFÁSICO CON TRANSFERENCIA DE CALOR EN UNIDADES DE PROCESAMIENTO GRÁFICO

Thomás Coronel

Mgter. Ezequiel Fogliatto
Director

Ing. Pablo Argañaras
Co-director

Miembros del Jurado
Dr. René Cejas Bolecek
Dr. Flavio Colavecchia

Junio de 2020

Departamento de Mecánica Computacional
Centro Atómico Bariloche

Instituto Balseiro
Universidad Nacional de Cuyo
Comisión Nacional de Energía Atómica
Argentina

Para
Norma
Carlos
Sonia
Leandro
Chichi
Kiki
.

Resumen

En este trabajo se implementó un código numérico para resolver problemas de mecánica de fluidos con transferencia de calor en flujos multifásicos con cambio de fase utilizando Unidades de Procesamiento Gráfico (GPU). Se utilizó la arquitectura de las GPU por su diseño, el cual permite realizar la ejecución de instrucciones con elevado nivel de paralelismo, y así reducir los costos computacionales de los algoritmos involucrados.

El modelo de lattice Boltzmann (LBM) pseudopotencial es el utilizado para abordar los problemas de interés, el cual resuelve de manera indirecta ecuaciones diferenciales no lineales por medio de ecuaciones lineales más sencilla, y además tiene la ventaja de resultar altamente paralelizable. En particular, para el presente trabajo se utilizó un LBM con dos ecuaciones pseudopotencial con operador MRT.

El código realizado se implementó en los lenguajes de programación C, Cuda C y Python, en dónde el desarrollo del proyecto se llevó a cabo mediante la herramienta CMake con la posibilidad de seleccionar su tipo de variables (*float* o *double*). El proyecto se planificó para que los *kernels* de Cuda C compilados puedan ser implementados en un *script* de Python con su módulo PyCuda.

La validación del código se realizó en CPU y GPU para diferentes problemas numéricos. Se realizó una comparación entre los tiempos de cálculo del código en los diferentes lenguajes utilizados, en donde se obtuvo que la implementación en Cuda C llega a ser 23 veces más rápida que el equivalente en C para un único proceso, en las GPU y CPU evaluadas durante este Proyecto Integrador. Por otro lado, al realizar la comparación del código implementado en PyCuda con respecto al equivalente en Cuda C, se obtuvo que el primero es alrededor de un 10 % más lento que el segundo. En cuanto a la comparación de la conveniencia de utilización de tipos de variable *float* o *double*, se obtuvo que la diferencia porcentual entre las precisiones es cercana al 0,003 %, y que el tiempo de cálculo en variables tipo *double* es aproximadamente un 25 % mayor que en *float*.

Palabras clave: LATTICE BOLTZMANN, FLUJO MULTIFÁSICO, TRANSFERENCIA DE CALOR, GPU

Abstract

In this work, a numerical code for multiphase flow with phase change and heat transfer was implemented in Graphics Processing Units (GPU). The GPU architecture is used by its design, which performs executions in parallel and thereby reduce the computational costs of the algorithms implemented.

A pseudopotential lattice Boltzmann model (LBM) was used, which indirectly recovers the solution of nonlinear differential equations from the solution of a much simpler equation with a highly paralellizable algorithm. In particular, a pseudopotential LBM with MRT operator was implemented.

Code was implemented under CMake on the programming languajes C, CUDA C and Python, with the posibility to select variable types at compilation time. CUDA C kernels were also compiled to be used within Python scripts by means of the PyCuda module.

Code validation was performed on CPU and GPU for different numerical benchmarks. A comparison was made between the calculation times of the code on the different programming languages, where it was obtained that the implementation on Cuda C improvement from one to two orders of magnitude to the equivalent implementation on C. On the other hand, when comparing the code implemented on PyCuda with respect to the equivalent on Cuda C, the former was found to be about 10% slower. Numerical solutions computed with different variable types (i.e. float or double) showed relative differences lower than 0.003%, while the computational time is increased by 25% between each case.

Keywords: LATTICE BOLTZMANN, MULTIPHASE FLOW, HEAT TRANSFER, GPU

Índice de contenidos

Índice de símbolos

CFD : Computational Fluid Dynamics

CPU : Central Processing Unit

EOS : Equation of State

GPU : Graphics Processing Unit

LBE : lattice Boltzmann equation

LBM : lattice Boltzmann Method

MRT : multiple relaxation times

SU : Speed Up

SD : Speed Down

VdW : Van der Waals

Capítulo 1

Introducción

En el estudio de la Mecánica de los Fluidos, son de importancia los problemas de transferencia de calor de flujos multifásicos con cambios de fase. Una de las áreas más relevantes en que se tratan estos problemas es la generación de energía eléctrica, ya sea por medio de fisión o fusión nuclear, la combustión de combustibles fósiles, fuentes de energía geotérmica entre otros [1]. Actualmente se estan investigando formas de eficientizar la producción de energía debido a que se espera un incremento de la demanda por la población [2].

Una de las ramas de investigación es en la industria nuclear, en donde se pretende mejorar la eficiencia en la remoción del calor de las barras del elemento combustible hacia el fluido del circuito de refrigeración primario. En particular, resulta de interés evaluar dicha transferencia de calor mediante procesos de ebullición en el fluido que entra en contacto con los elementos combustibles. Sin embargo, la realización de experimentos resulta costosa debido a los recaudos de seguridad y equipos necesarios. Estas limitaciones intrínsecas motivan el desarrollo de modelos numéricos capaces de predecir adecuadamente los fenómenos involucrados [3].

Actualmente la dificultad para reproducir numéricamente los tipos de problemas mencionados se debe a las escala de fluido involucrada (*mesoscópica*), con las técnicas numéricas convencionales de Mecánica de Fluidos Computacional (*Computational Fluid Mechanichs* o CFD) resulta difícil obtener resultados en un tiempo razonable y que también tengan en cuenta todas las interacciones físicas que ocurren [4]. En parte es causado por los métodos tradicionales que resuelven ecuaciones diferenciales mediante técnicas de discretización que involucran la resolución numérica de grandes sistemas de ecuaciones, donde el tiempo de cálculo típicamente varía cuadráticamente con el número de elementos de malla utilizada para discretizar el problema físico.

Uno de los métodos numéricos que se están desarrollando para que el costo computacional sea cada vez menor es el método de lattice Boltzmann (*lattice Boltzmann Method* o LBM). Estos métodos resuelven las ecuaciones diferenciales por medio de una ecuación

lineal más sencilla, lo cuál reduce drásticamente el tiempo de cálculo, y las variables que se obtienen cumplen con la ecuación de interés. Debido a que están basados en realizar operaciones elementales en los elementos de la malla discretizada y a su vez no dependen del total de nodos de la misma, se convierten en algoritmos altamente paralelizables.

Por lo mencionado anteriormente, el estudio de la transferencia de calor en flujos multifásicos con cambio de fase es difícil de modelar numéricamente y los métodos existentes son costosos computacionalmente. Esto implica que resulta necesario desarrollar e investigar nuevas técnicas numéricas junto con sus implementaciones, de modo que sea posible resolver la fenomenología de interés con precisión y en tiempo razonable.

1.1. Descripción general de LBM

Tradicionalmente los problemas de mecánica de fluidos resueltos mediante la suposición del continuo son llevados a cabo mediante ecuaciones como las de Navier-Stokes. Obtener la solución numérica de dichas ecuaciones implica resolver sistemas algebraicos de ecuaciones, cuya cantidad está sujeta a la forma de realizar la discretización del espacio físico y según la cantidad de elementos de la malla. La resolución de este tipo de ecuaciones presenta la característica de que el tiempo de cálculo involucrado en su resolución depende típicamente de forma cuadrática con la cantidad de elementos discretizados [5].

El LBM resuelve de manera indirecta las ecuaciones diferenciales, a través de la discretización de una ecuación de Boltzmann, lineal y de resolución numérica más sencilla. En particular, dichas ecuaciones representan el transporte de funciones de distribución, cuyos momentos en el espacio de fases pueden asociarse con propiedades macroscópicas del fluido, como densidad, velocidad o temperatura.

La discretización espacial que se suele utilizar es mediante un mallado regular [4], y cada nodo de la malla tiene asignado un espacio de velocidades. Los modelos de grilla suelen denotarse como $DdQq$, donde d son las dimensiones y q la cantidad de componentes que se discretiza el espacio de velocidades. La Figura (1.1) muestra cómo son los conjuntos de velocidades para los modelos D1Q3 y D1Q9.

El espacio de velocidades indica cómo es la propagación de las propiedades que poseen los nodos de la grilla. La velocidad de grilla del nodo i-ésimo se denota $\mathbf{e}_i$ y posee q componentes. Para el modelo D2Q9 la Figura 1.1 muestra un esquema de las velocidades de grilla del nodo i-ésimo y la Ec. (1.1) el valor adoptado.

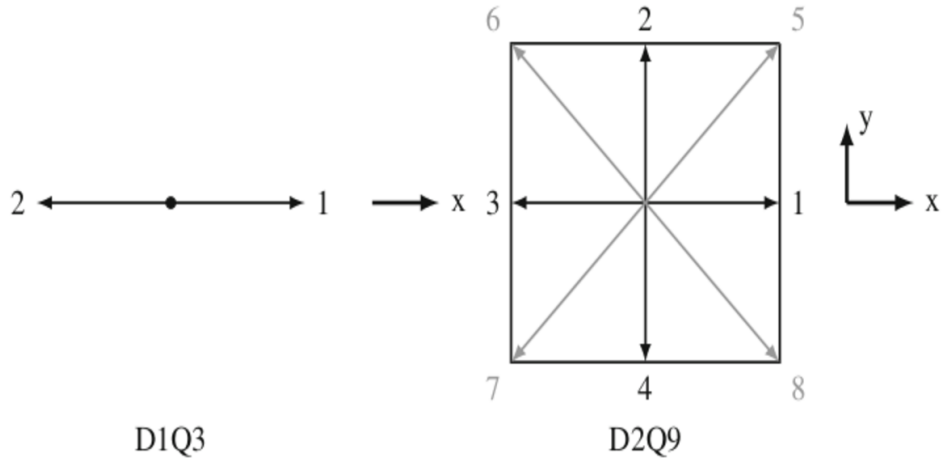


Figura 1.1: Conjunto de velocidades de los modelos D1Q3 y D2Q9. [6]

$$\mathbf{e}_i = \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{pmatrix} = \begin{pmatrix} (0,0,0) \\ (1,0,0) \\ (0,1,0) \\ (-1,0,0) \\ (0,-1,0) \\ (1,1,0) \\ (-1,1,0) \\ (-1,-1,0) \\ (1,-1,0) \end{pmatrix} \tag{1.1}$$

Para la resolución de los problems de transferencia de calor con flujos multifásicos y cambio de fase que se propuso estudiar, es necesario contar con un LBM adecuado. En particular, existen cuatro (4) categorías generales para clasificar los modelos multifásicos: *color gradient*, *free energy*, *phase field* y *pseudopotential*.

En este trabajo se utilizará la familia pseudopotencial, la cual está basada en proponer un potencial de interacción entre las partículas del fluido. Dicho potencial se utiliza para calcular la fuerza de interacción entre las partículas del fluido y está dado según una Ecuación de estado (*Equation of state o EOS*).

A continuación se datallará un ejemplo sencillo de un modelo D2Q9. La Figura (1.2) muestra un esquema de las direcciones de las velocidades de grilla del nodo i-ésimo. En la misma figura se esquematiza cómo es el proceso de colisión y advección (*streaming*) que se realiza para actualizar los estados de los nodos cuando transcurre un paso de tiempo de la discretización temporal del problema.

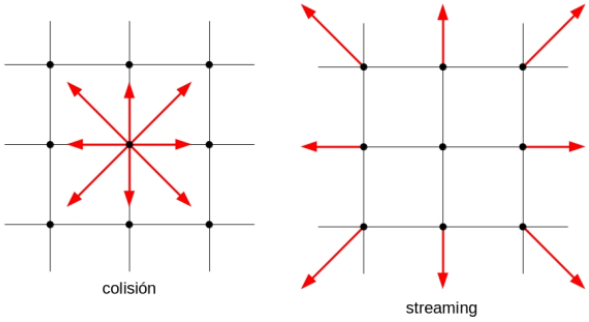


Figura 1.2: Colisión y streaming de un modelo D2Q9 de LBM.

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \delta_t, t + \delta_t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha{}^{eq}) \tag{1.2}$$

El término derecho de la Ec. (1.2) se conoce como *colisión* y el izquierdo como *streaming*; α corresponde al α-ésimo componente ($\alpha = 0, 1, ..., 8$), $\mathbf{f^{eq}}$ es la función de distribución en estado de equilibrio, τ es un parámetro de relajación del modelo que determina propiedades macroscópicas como la viscosidad.

Por medio de la *colisión* y luego del *streaming* se obtienen los parámetros macroscópicos del problema. Para este caso sencillo se obtiene ρ y $\mathbf{u}$ de las Ec. (1.3) y (1.4) respectivamente.

$$\rho = \sum_\alpha f_\alpha \tag{1.3}$$

$$\rho \mathbf{u} = \sum_\alpha \mathbf{e}_\alpha f_\alpha \tag{1.4}$$

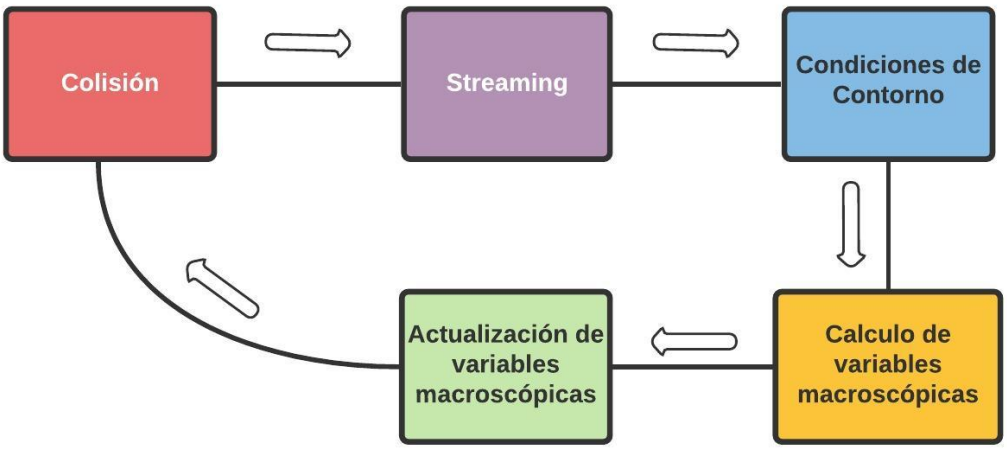


Figura 1.3: Esquema de resolución de un método numérico de LB.

La Figura (1.3) muestra el esquema de resolución para el LBM descripto, donde primero se realiza la colisión de los nodos para luego mediante el *streaming* actualizar en todos los nodos el valor de $\mathbf{f}$. A partir de ahí se deben aplicar las condiciones de contorno que pueda tener el problema y calcular las variables macroscópicas del sistema. Una vez realizada la actualización de las variables macroscópicas se recupera la solución de la ecuación diferencial de interés. El proceso descripto debe aplicarse en cada paso de tiempo.

Debido a que cada nodo de la grilla debe realizar la misma operación de colisión de manera independiente del resto, el modelo es altamente paralelizable. Además, las operaciones matemáticas que deben ejecutarse en el operador de colisión son sencillas, no implicando un gran costo computacional.

Por lo que se describió anteriormente, el LBM permite resolver indirectamente una ecuación, o EDP no lineal a partir de una ecuación lineal más sencilla y es paralelizable. Por lo tanto, esto motiva el desarrollo de códigos numéricos en unidades de procesamiento gráfico (*Graphics Processing Unit* o GPU).

1.2. Descripción GPU

Una Unidad de Procesamiento Gráfico (*Graphics Processing Unit* o GPU) es un circuito electrónico diseñado para realizar operaciones con punto flotante para renderizar píxeles en una pantalla. Están optimizadas para actuar en paralelo de forma simultánea instrucciones simples.

La GPU trabaja en conjunto con una Unidad Central de Procesamiento (*Central Processing Unit* o CPU), debido a ello no presenta circuitos de control en su arquitectura. Dicho espacio que se encuentra ocupado en las CPU por el circuito de control las GPU lo disponen para incrementar el espacio en su chip de Unidades Aritmético Lógicas (*Arithmetic Logic Unit* o ALU).

El esquema de la Figura (1.4) muestra cualitativamente la cantidad de transistores dedicados a diferentes tareas en la CPU comparado con la GPU.

Figura 1.4: Comparación cualitativa del uso de transistores entre CPU y GPU [7].

Las GPUs por como es su arquitectura de diseño estan preparadas para ejecutar cálculos en paralelo, como los que presentan los métodos numéricos de LB. Los LBM consisten en que a cada uno de los nodos de la grilla discretizada se le realicen operaciones elementales como los de la Ec. (1.2). Las GPU se encuentran optimizadas para ejecutar cálculos sobre los píxeles de un monitor, los cuales pueden considerarse como una grilla de nodos. A través de la paralelización la performance de las GPUs es alta, siendo capaces de procesar múltiples vértices y píxeles simultáneamente. Las simulaciones numéricas se producen en un menor tiempo en las GPU que sobre las CPU debido al alto paralelismo alcanzado por la GPU [7].

El lenguaje de programación usado para trabajar con placas gráficas fue CUDA C y lo desarrolló la empresea NVIDIA. Se basa en el lenguaje de programación C con ciertas modificaciones para que los procesos sean en paralelo. Dichos procesos se especifican para que puedan ser lanzados en un número de *blocks* y *threads* de ejecución.

1.3. Descripción y alcance del proyecto integrador

En el presente trabajo se implementó un código numérico para resolver problemas de transferencia de calor en flujos multifásicos con cambio de fase usando GPU de la forma más robusta posible. El LBM utilizado es el desarrollado por Fogliatto et al [8], [9], [10], y resuelve numéricamente problemas con discretización espacial de dominio regular en una grilla de tipo D2Q9.

El proceso de desarrollo del software fue realizado con la herramienta GIT, un sistema de control de versiones distribuido que se emplea para administrar los flujos de trabajo del desarrollo de software. El código se encuentra implementado en tres lenguajes de programación: C, CUDA C y PYTHON. Las bibliotecas generadas son del tipo *shared* y *static* para el caso de C y CUDA C respectivamente. Adicionalmente los *kernels* de CUDA C se compilaron en el formato tipo PTX para ser utilizadas mediante PYTHON por medio del módulo PYCUDA. Para administrar el proceso de compilación del software se empleó el software CMAKE, que permite desarrollar proyectos con jerarquías de directorios y aplicaciones que dependen de múltiples bibliotecas.

La validación del código fue realizado en dos PC diferentes, la primera contaba con una CPU Intel Core i7-3770 con una GPU NVIDIA GeForce GTX 760 y la segunda con una CPU Intel Core i7-4770 con una GPU NVIDIA GeForce GTX 970. A su vez la validación se concretó en variables de simple y doble precisión, a través de la resolución de diferentes problemas numéricos (detallados en el Cap. (4)):

- *Construcción de Maxwell*
- *Estratificación de un fluido Van der Waals (VdW) con temperatura no uniforme*
- *Generación de burbujas sobre una superficie horizontal calefaccionada*

Se realizó una comparación de tiempos de cálculo entre los códigos de C y en CUDA C para ambas precisiones. Luego se realizó una comparación entre los tiempos de cálculo de simple precisión y doble precisión en el código de CUDA C.

Además, se usaron *kernels* compilados en formato PTX de CUDA C dentro de *scripts* de PYTHON usando PYCUDA, y se realizó una comparación de tiempos de cálculo entre CUDA C y PYTHON.

Capítulo 2

Modelo de lattice Boltzmann

2.1. LBM Multifásicos

En este capítulo se presenta el modelo de lattice Boltzmann (LBM) que será utilizado en la resolución de problemas con flujo multifásico y transferencia de calor, mediante una implementación en unidades de procesamiento gráfico.

Típicamente, las ecuaciones que describen la mecánica de fluidos suelen ser de difícil resolución y las soluciones analíticas de problemas que pueden ser halladas son escasas, como el caso de flujos *Couette* o *Poiseuille*. Es difícil encontrar la solución de las ecuaciones de la mecánica de fluidos en problemas con geometrías complejas u otras condiciones de contorno, si es que el problema la tiene. Debido a ello las soluciones se obtienen numéricamente [6], por lo que es de importancia el desarrollo de métodos numéricos que resuelvan los problemas de forma paralela y por medio de este proceso, reducir el tiempo de cálculo.

Los problemas que se plantean resolver en el presente trabajo son de transferencia de calor en flujos multifásicos con cambio de fase y la escala de fluido adoptada es la mesoscópica. Considerando la escala y siendo un problema multifásico, se opta por resolver numéricamente mediante LBM.

La mayoría de los modelos LBM para resolver los flujos multifásicos son clasificados en cuatro categorias generales: *color gradient*, *Shan Chen model* o *pseudopotential*, *free-energy* y *phase-field*.

- *Color gradient*: fue el primer modelo de LBM para flujos multifásicos desarrollado por Gunstensen [11]. Las fases y las interacciones entre las partículas son denotadas mediante diferentes colores. Por medio del modelado local del gradiente de color que se encuentra asociado a la diferencia de las densidades de las dos fases, se conoce cómo es la segregación y separación de las fases.

- *Shan Chen model* o *pseudopotential* surge de representar la fenomenología de *color gradient* por medio de una redistribución de las partículas del fluido. La fuerza de interacción proviene de la diferencia entre las fuerzas promedio del modelo molecular entre ambos lados de la interface. Shan and Chen [12] presentaron un modelo de LBE (referenciado como modelo SC) que podría representar la interacción entre partículas fluidas de forma más precisa y directa introduciendo un pseudo-potencial. En este caso, el pseudo-potencial que produce la fuerza de interacción entre partículas fluidas está basado en la incorporación de una Ecuación de estado (EOS) que permita la coexistencia de fases en equilibrio.

- *Free-energy* es un tipo de modelo alternativo de LBM desarrollado por Swift [13] para modelos multifásicos/multicomponentes basado en la teoría de energía libre (*free-energy*). La idea básica del nuevo método es realizar una función de distribución de equilibrio basada en funciones de energía libre, en las cuales se incorpora el tensor de presión termodinámico [4].

- *Phase-field* utiliza en su modelo un parámetro gobernado por una ecuación de tipo convección-difusión para realizar un seguimiento de la interfase. Presenta una gran estabilidad numérica y precisión para problemas con grandes relaciones de densidad y viscosidad [14].

2.2. Modelo pseudopotencial

El modelo pseudopotencial es el adoptado en el presente trabajo, donde dicho modelo resuelve problemas cuya discretización espacial está realizada de forma regular. Este modelo multifásico tiene la particularidad de que la frontera entre las fases no es resuelta con exactitud, sino que dicha interfase es representada de forma difusa con un cierto tamaño en la grilla, siendo una importante ventaja para el cálculo puesto que la interfase no debe ser reconstruida [15].

La obtención de la ecuación de lattice Boltzmann (LBE) a través de la ecuación de Boltzmann se encuentra descripta con detalle en [6], de modo que sólo se incluirá una breve descripción. El problema físico a resolver cuenta con una región a la que se le realizará un mallado regular para discretizar el espacio. El nodo i-ésimo de la malla posee las coordenadas $\mathbf{X}_i = (x, y, z)$, a su vez densidad ρ_i y temperatura T_i. La velocidad del fluido en el nodo tiene las componentes $\mathbf{U}_i = (U_{ix}, U_{iy}, U_{iz})$. El espacio de velocidades indica la propagación de las propiedades en la grilla, donde la velocidad de grilla $\mathbf{e}_i$ posee q componentes.

Los modelos de grilla suelen denotarse como $DdQq$, donde d son las dimensiones del espacio euclídeo y q la cantidad de componentes en que se discretiza el espacio de fases. En el presente trabajo el modelo de grilla empleado es D2Q9, donde el esquema

de velocidades de grilla para el nodo i-ésimo del D2Q9 es el presentado en la Figura (1.1). La Ec. (1.1) detalla los valores de velocidad adoptados.

En el caso más simple se cuenta con una única función de distribución **f**, y sus primeros momentos discretos en el espacio de velocidades están asociados a propiedades macroscópicas del fluido, como la densidad y velocidad del fluido.

En este modelo es necesario porporcionar un potencial que describa a las fuerzas de interacción entre los nodos y dicho potencial estará dado por una Ecuación de estado (*Equation of state* o EOS). Es de importancia la elección de la EOS a utilizar, puesto que según ella se recupera el tensor de presión y las variables ρ, U, T; a una dada presión y temperatura quedan determinadas las densidades de coexistencia de líquido y gas así como el calor latente en entre otras propiedades intrínsecas del fluido.

2.3. Ecuación de estado y fluido de Van der Waals

Al ser multifásicos los problemas a desarrollar, es de importancia conocer las leyes que describen la separación de fases. La ley de gases ideales Ec. (2.1) es un ejemplo de EOS , donde p es la presión (atm), V es el volumen (L), n número de moles, R constante universal de los gases y T temperatura. La ley caracteriza el comportamiento para gases de baja densidad.

$$pV = nRT \tag{2.1}$$

La EOS de Van der Waals (VdW) Ec.(2.2) fue propuesta para caracterizar el comportamiento de los gases reales, siendo $V_m = \frac{V}{n}$ el volumen molar. Las constantes a y b son características de cada fluido.

$$p = \frac{RT}{V_m - B} - A\left(\frac{1}{V_m}\right)^2 \tag{2.2}$$

El parámetro A ($\frac{atmL^2}{mol^2}$) caracteriza la interacción que poseen las moléculas del gas entre sí, y B ($\frac{L}{mol}$) da una idea del volumen molar mínimo que posee una partícula del fluido (este parámetro define a la partícula del gas con un dado volumen en vez de ser puntual como en la Ley de gases ideales).

La EOS de VdW es utilizada en el presente trabajo para modelar el potencial del modelo LBM *pseudopotential* de interacción entre las partículas. Su elección permite la coexistencia de diferentes densidades para una misma condición de temperatura y presión [16].

2.4. Modelo pseudopotencial de dos ecuaciones con operador MRT

En esta sección se describirá cuál es el LBM utilizado para resolver problemas con discretización espacial de dominio regular. Se deben resolver dos ecuaciones, la primera es la hidrodinámica que representa a la conservación de masa y momento; la segunda a la de energía, descriptas en [10].

2.4.1. Ecuación hidrodinámica

Por medio de la resolución de una LBE para la función de distribución $\mathbf{f}$ (Ec.(2.3)) definida en el espacio de poblaciones (*distribution function in population space*), se puede obtener la solución de las ecuaciones hidrodinámicas [17]:

$$\mathbf{f}(\mathbf{x}+\mathbf{e}\,\delta_t, t+\delta_t) = \mathbf{M}^{-1}\left[\mathbf{m} - \mathbf{\Lambda}(\mathbf{m}-\mathbf{m^{eq}}) + \delta_t\left(\mathbf{I}-\frac{1}{2}\mathbf{\Lambda}\right)\bar{\mathbf{S}}\right]_{(\mathbf{x},t)} \tag{2.3}$$

donde $\mathbf{x}$ es la posición espacial, t el tiempo, δ_t el paso de tiempo, $\boldsymbol{e}$ la velocidad de grilla en sus direcciones α y $\mathbf{f}$ es la función de distribución de densidad definida en el espacio de poblaciones. La notación utilizada implica que la componente α-ésima del miembro izquiero de la Ec. (2.3) esté dado por $f_\alpha(x + e_\alpha\delta_t, t + \delta_t)$. El miembro derecho de la Ec. (2.3) corresponde a la etapa de post-colisión definida en el espacio de momentos, donde $\mathbf{I}$ es el tensor identidad, $\mathbf{M}$ una matriz de transformación ortogonal, $\mathbf{m} = \mathbf{M}\cdot\mathbf{f}$, $\mathbf{m^{eq}} = \mathbf{M}\cdot\mathbf{f}^{\mathbf{eq}}$, $\bar{\mathbf{S}} = \mathbf{MS}$ el término de fuente y $\mathbf{\Lambda}$ es una matriz diagonal que depende del modelo *DdQq*, $\mathbf{f}$, y coeficientes como la viscocidad y difusividad térmica [18]. Este tipo de operador se conoce como MRT (*Multiple Relaxation Times*), y para este modelo quedan definidos como:

$$\mathbf{m^{eq}} = \rho\left(1, -2+3|\mathbf{u}|^2, 1-3|\mathbf{u}|^2, u_x, -u_x, u_y, -u_y, {u_x}^2 - {u_y}^2, u_x\,u_y\right) \tag{2.4}$$

$$\mathbf{\Lambda} = diag({\tau_\rho}^{-1}, {\tau_e}^{-1}, {\tau_\zeta}^{-1}, {\tau_j}^{-1}, {\tau_q}^{-1}, {\tau_j}^{-1}, {\tau_q}^{-1}, {\tau_\nu}^{-1}, {\tau_\nu}^{-1}) \tag{2.5}$$

Cada modelo *DqDq* está asociado a un set de velocidades de grilla para discretizar el espacio de fases, lo cual fija los coeficientes de la la matriz de transformación $\mathbf{M}$. En particular, para el modelo D2Q9 esta matriz está determinada por [6]:

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & -1 & -1 & -1 & -1 & 2 & 2 & 2 & 2 \\ 4 & -2 & -2 & -2 & -2 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & -2 & 0 & 2 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \\ 0 & 0 & -2 & 0 & 2 & 1 & 1 & -1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \end{bmatrix} \tag{2.6}$$

La densidad macroscópica es obtenida mediante :

$$\rho = \sum_{\alpha} f_{\alpha} \tag{2.7}$$

Por medio de la Ec. (2.8) se obtiene la velocidad.

$$\rho\, \mathbf{u} = \sum_{\alpha} \mathbf{e}_{\alpha}\, f_{\alpha} + \frac{1}{2}\, \delta t\, \mathbf{F} \tag{2.8}$$

Para el modelo D2Q9 la fuerza total $\mathbf{F}$ posee sólo dos componentes F_x, F_y. Donde $\mathbf{F} = \mathbf{F}_b + \mathbf{F}_{int}$, siendo $\mathbf{F}_b$ la fuerza volumétrica y $\mathbf{F}_{int}$ la fuerza de interacción que hay en el sistema dadas por las Ecs. (2.9) y (2.10) respectivamente.

$$\mathbf{F}_b = \rho\, \mathbf{g} \tag{2.9}$$

$$\mathbf{F}_{int} = -G\, \psi(\mathbf{x}) \sum_{\alpha} w(|\mathbf{e}_{\alpha}|^2)\, \psi(\mathbf{x} + \mathbf{e}_{\alpha}\, \delta_t)\, \mathbf{e}_{\alpha} \tag{2.10}$$

Donde G corresponde a la intensidad de interacción, $w(|\mathbf{e}_{\alpha}|^2)$ son los pesos correspondientes a una grilla D2Q9 y ψ es el potencial :

$$\psi(\rho) = \sqrt{\frac{2(p_{EOS} - \rho\, {c_s}^2)}{G c^2}} \tag{2.11}$$

donde c_s es la velocidad del sonido en el medio. La EOS adoptada en el presente trabajo es la de VdW :

$$p_{EOS} = \frac{\rho R T}{1 - \rho\, b} - a\rho^2 \tag{2.12}$$

siendo a y b parámetros que determinan los valores críticos de temperatura, presión y densidad. Finalmente, la fuerza de interacción se incorpora en la etapa de colisión mediante un término de fuente apropiado:

$$\bar{S} = \begin{bmatrix} 0 \\ 6\mathbf{u}\cdot\mathbf{F} + \frac{12\sigma|\mathbf{F}_{int}|^2}{\psi^2\delta_t(\tau_e - 0{,}5)} \\ 6\mathbf{u}\cdot\mathbf{F} - \frac{12\sigma|\mathbf{F}_{int}|^2}{\psi^2\delta_t(\tau_\zeta - 0{,}5)} \\ F_x \\ -F_x \\ F_y \\ -F_y \\ 2(u_xF_x - u_yF_y) \\ (u_xF_x + u_yF_y) \end{bmatrix} \tag{2.13}$$

donde σ es un parámetro libre del modelo MRT, al igual que Λ, el cual se utiliza para ajustar las diferencias de fases obtenidas de la EOS y de la simulación realizada.

Se pueden recuperar las ecuaciones de Navier - Stokes utilizando Ecs. [2.7 - 2.13] mediante el análisis de Chapman-Enskog limitado para un número de Mach bajo [17], donde las ecuaciones recuperadas son [9] [17]:

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{u}) = 0 \tag{2.14}$$

$$\frac{\partial(\rho\mathbf{u})}{\partial t} + \nabla\cdot(\rho\mathbf{u}\mathbf{u}) = -\nabla\left(\rho {c_s}^2\mathbf{I}\right) + \nabla\mathbf{\Pi} + \mathbf{F} - 2G^2c^4\sigma\nabla\left(|\nabla\psi|^2 I\right) + O(\partial^5) \tag{2.15}$$

$$\mathbf{F} = -Gc^2\left[\psi\nabla\psi + \frac{1}{6}c^2\psi\nabla\left(\nabla^2\psi\right) + ...\right] + \mathbf{F}_b = \mathbf{F}_{int} + \mathbf{F}_b \tag{2.16}$$

En éste caso, el tensor de viscosidad $\mathbf{\Pi}$ queda determinado por :

$$\mathbf{\Pi} = \rho\nu\left[\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right] + \rho(\,\xi - \nu)\,(\nabla\mathbf{u})\,\mathbf{I} \tag{2.17}$$

siendo $\nu = {c_s}^2(\tau_\nu - 0,5)\delta_t$ la viscosidad cinemática y $\xi = {c_s}^2(\tau_\nu - 0,5)\delta_t$ la viscosidad volumétrica.

2.4.2. Ecuación de energía

Para tener en cuenta la transferencia de calor en el modelo, es necesario adicionar otra ecuación de LB acoplándola con la Ec. (2.3) [17]. Para nuestro caso, se utiliza la distribución de poblaciones $\mathbf{g}$ en la Ec. (2.18), la cual también posee un operador de colisión MRT, donde $\mathbf{n} = \mathbf{Mg}$ es una distribución de momentos (no confundir con el número de moles definido anteriormente) y $\hat{\mathbf{\Gamma}}$ es una fuente en el espacio de momentos.

$$\mathbf{g}(\mathbf{x}+\mathbf{e}\delta_t, t+\delta_t) = \mathbf{M}^{-1}\left[\mathbf{n} - \mathbf{Q}(\mathbf{n}-\mathbf{n}^{eq}) + \delta_t\left(I - 0,5Q\right)\hat{\Gamma}\right]_{(\mathbf{x},t)} \tag{2.18}$$

En este caso los parámetros libres del modelo MRT vienen dados en parte por la matriz de coeficientes de relajación $\mathbf{Q}$, siendo compuesta por la diagonal que se indica en Ec. (2.19) y además por los elementos no nulos $Q_{3,4}$ y $Q_{5,6}$ que se indican en las Ecs. (2.20) y (2.21) respectivamente.

$$diag(Q) = (q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8)^T \tag{2.19}$$

$$Q_{3,4} = q_4\left(\frac{q_3}{2} - 1\right) \tag{2.20}$$

$$Q_{5,6} = q_6\left(\frac{q_5}{2} - 1\right) \tag{2.21}$$

La distribución de equilibrio $\mathbf{n^{eq}}$ se encuentra definida en Ec.(2.22), siendo α_1 y α_2 parámetros libres del modelo :

$$\mathbf{n^{eq}} = T(1, \alpha_1, \alpha_2, u_x, -u_x, u_y, -u_y, 0, 0)^T \tag{2.22}$$

La temperatura macroscópica T puede recuperarse mediante:

$$T = \sum_\alpha g_\alpha + \frac{1}{2}\delta_t\hat{\Gamma}_0 \tag{2.23}$$

donde el término fuente resulta :

$$\hat{\mathbf{\Gamma}} = (s, 0, 0, 0, 0, 0, 0, 0, 0)^T \tag{2.24}$$

con

$$s = \frac{\chi}{\rho}\nabla T\cdot\nabla\rho + T\left[1 - \frac{1}{\rho c_\nu}\left(\frac{\partial p_{EOS}}{\partial T}\right)_\rho\right]\nabla\cdot\mathbf{u} \tag{2.25}$$

Por medio de las Ecs. (2.18) y (2.23) se recupera la ecuación del calor [19]:

$$\frac{\partial T}{\partial t} + \nabla\cdot(\mathbf{u}T) = \chi\,\nabla^2 T + s \tag{2.26}$$

donde χ es la difusividad térmica:

$$\chi = \delta_t\left(\frac{1}{q_3} - \frac{1}{2}\right)\left(\frac{4+3\alpha_1+2\alpha_2}{6}\right) \tag{2.27}$$

En el presente trabajo, se considera χ constante en cada fase, y puede verse que queda determinada por coeficientes de relajación específicos y los parámetros libres α_1 y α_2.

2.4.3. Condiciones de contorno

Cuando se realiza el *streaming* en los nodos que se encuentran en la frontera de la región a resolver, no se poseen todos los valores de la función de distribución de poblaciones **f** y **g**, por lo que es necesario determinar cómo se obtienen los mismos.

El código numérico que se desarrolló es para dominios cuadrados o rectangulares con condiciones de contorno periódicas. En este caso, las fronteras a analizar corresponden a una de las cuatro (4) aristas del dominio. Como la resolución es idéntica, sólo se detalla el procedimiento para una.

La Figura (2.1) ilustra las direcciones de las componentes de **f** y **g** para un nodo que se encuentra en la frontera. La disposición observada es la que se evaluará para las condiciones de contorno hidrodinámica y de energía. Por convención las direcciones para las que se conocen los valores de las funciones de distribución pertenecen al conjunto A y las que no se conocen al conjunto B. Por lo que $0, 1, 2, 4, 5, 8 \in A$ y $3, 6, 7 \in B$.

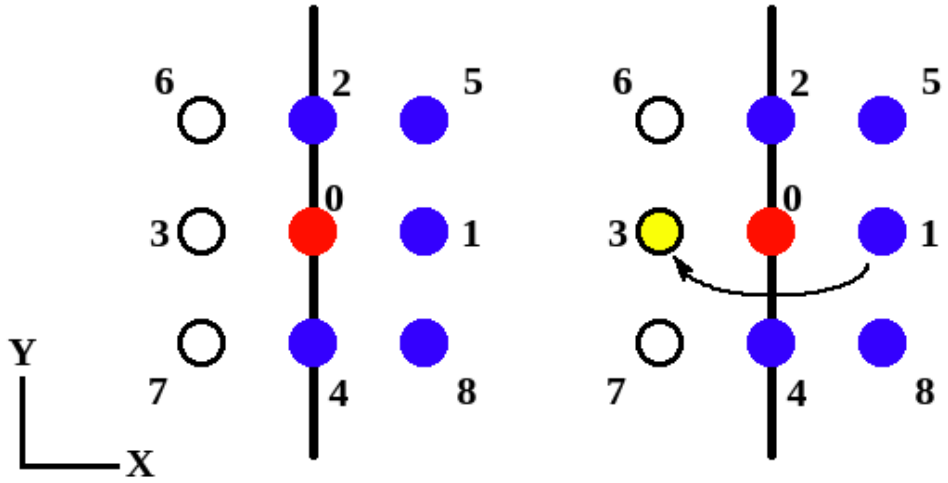


Figura 2.1: Izquierda: Direcciones de las componentes de las funciones de distribución de poblaciones **f** y **g**, para un nodo que se encuentra en la frontera. Derecha: método de *bounce-back* aplicado a la dirección 3.

Condición hidrodinámica

El desarrollo de Zou [20] es el utilizado para la ecuación hidrodinámica. En primer lugar se aplica el método *bounce-back* a la dirección 3, resultando $f_3 = f_1$ como se observa en el panel derecho de la Figura (2.1). Luego se analizan las Ecs. (2.7) y (2.8) que son puestas de la siguiente forma:

$$\begin{aligned} f_3 + f_6 + f_7 &= \rho - (f_0 + f_1 + f_2 + f_4 + f_5 + f_8) \\ f_6 - f_7 &= \rho u_x - (f_1 - f_3 - f_7 + f_8) \\ f_3 + f_6 + f_7 &= \rho u_y + (f_1 + f_5 + f_8) \end{aligned} \tag{2.28}$$

Para finalizar se aplican las condiciones de no deslizamiento, en éste caso $u_y = 0$ y $\mu\frac{\partial u}{\partial x} = \tau_{wall}$, siendo τ_{wall} el esfuerzo de corte realizado en la arista analizada. Se obtienenen las siguientes condiciones de contorno a la distribución de poblaciones $\mathbf{f}$:

$$\begin{aligned} f_3 &= f_1 \\ f_7 &= f_5 - 0,5(f_4 - f_2) - 0,25(F_x + F_y) \\ f_6 &= f_8 + 0,5(f_4 - f_2) - 0,25(F_x + F_y) \end{aligned} \tag{2.29}$$

en cada nodo de la frontera, donde F_x y F_y son las componentes de la fuerza que posee el nodo.

Para obtener una resolución de problemas genéricos, se debe agregar la condición de contorno en el caso de tener vértices en la región. Es este tipo de nodos, la cantidad de componentes a resolver origina problemas sub o sobredeterminados, dependiendo de la concavidad del dominio en el punto bajo análisis. Por lo tanto, es necesario aplicar condiciones adicionales a las descriptas para conservar adecuadamente las variables macroscópicas de interés.

Condición de energía

La condición de contorno a realizar en la ecuación de energía, consiste en mantener una temperatura fija en una de las aristas o en secciones de la misma. Para este caso se utiliza el desarrollo de Inamuro [21].

Primero se calcula el valor de la función de distribución definida en el espacio de poblaciones en equilibrio del nodo i-ésimo $\mathbf{g^{eq}}$ utilizando Ec. (2.22) y $\mathbf{M}$ de la forma $\mathbf{g^{eq}} = \mathbf{M}^{-1}\mathbf{n^{eq}}$. Se prosigue a calcular el parámetro β:

$$\beta = \frac{T_{cc} - \sum_A g_A}{\sum_B g_B} \tag{2.30}$$

donde T_{cc} es la temperatura fijada como condición de contorno, g_A adopta los valores conocidos de las direcciones pertenecientes al conjunto A $(0, 1, 2, 4, 5, 8)$, mientras que g_B adopta los valores de la dirección de $\mathbf{g^{eq}}$ calculada recientemente para las direcciones de B $(3, 6, 7)$.

Por último en las direcciones del conjunto B, el valor de g resulta:

$$g_B = \beta\, g_B^{eq} \tag{2.31}$$

A modo de ejemplo se detalla el valor que adoptará la dirección **3**:

$$g_3 = \frac{T_{cc} - \overbrace{(g_0 + g_1 + g_2 + g_4 + g_5 + g_8)}^{\sum_A g_A}}{\underbrace{(g_3^{eq} + g_6^{eq} + g_7^{eq})}_{\sum_B g_B}} \quad g_3^{eq} \tag{2.32}$$

Capítulo 3

Código numérico de LBM

En el presente capítulo se realizará la descripción de la implementación del código numérico del LBM descripto en la Sec. (2.4), como también las implicancias de elaborar la implementación en una GPU de forma eficiente.

El lenguaje de programación C desarrollado por Dennis MacAlistair Ritchie será utilizado para desarrollar el código. Dicho lenguaje brinda instrucciones a la CPU de una PC para ser ejecutadas. Las CPU son diseñadas óptimamente para que sus instrucciones sean procesadas de forma secuencial en los núcleos que poseen; aunque también se permite realizar los procesos en paralelo, según la cantidad de núcleos.

Luego se implementará un código en CUDA C, desarrollado por la empresa NVIDIA. Este lenguaje permite ejecutar instrucciones en una GPU, la cual está diseñada para que los procesos a realizar sean de forma paralela.

Se eligió la programación en C y CUDA C para comparar la eficiencia en el tiempo de cálculo, debido a que el lenguaje CUDA C es una extensión del lenguaje C. La diferencia principal es que CUDA C tiene una forma particular de escribir las funciones que se ejecutarán en la GPU, las cuales son llamadas *kernel*.

Ambos códigos, en C y CUDA C, son compilados mediante CMAKE en bibliotecas tipo *shared* y *static* respectivamente. También se realizó la compilación en formato PTX de los *kernels* de CUDA C, donde este formato puede utilizarse en el lenguaje de programación interpretado PYTHON. El módulo de PYTHON necesario para adquirir los *kernels* es PYCUDA.

Se utilizó PYTHON por la versatilidad que presenta para resolver problemas diversos. Esto se debe a que es un lenguaje simple, orientado a objetos y permite incorporar las bibliotecas compiladas en otros lenguajes. Además, PYTHON puede ser utilizado en distintos sistemas oparativos como *Linux*, *Windows* y *Mac OS*.

Para visualizar los resultados del código se utiliza el programa PARAVIEW, de modo que el formato que se eligió para la escritura de los campos es ENSIGHT GOLD. A través del lenguaje C, es que se hicieron las funciones de escritura de este tipo de archivos

y así observar gráficamente las variables de interés de los problemas que se desean resolver. El módulo CTypes de Python es el necesario para adquirir, de la biblioteca compilada en C, las funciones de escritura en formato Ensight Gold.

3.1. Programación en GPU

Una computadora (*Personal Computer* o PC) posee como procesador principal la CPU, cuyo diseño se encuentra optimizado para realizar tareas secuenciales. Comercialmente vienen de una amplia variedad de núcleos, en el rango de 8 a 64 núcleos como en el caso de los Procesadores AMD Ryzen™ Threadripper o 4 a 8 núcleos en los Procesador AMD FX™. En el caso de Intel se encuentra el Procesador Intel® Core™ serie X con 18 núcleos y Intel® Core™ I7-3770 de 4 núcleos entre otros [22] [23].

Los procesadores multinúcleo (hardware de multiprocesamiento), como los descriptos en el párrafo anterior, permiten ejecutar instrucciones en cada núcleo de forma independiente mediante la programación de subprocesos (*threads*) [24]. Esto permite obtener una mayor eficiencia en la ejecución de los procesos.

A su vez una PC opcionalmente puede contener un coprocesador siendo una GPU. Dicha placa se encuentra diseñada para realizar operaciones en paralelo, realizándolas en varios hilos de ejecución (*threads*).

El procesador principal que tiene una PC es la CPU, por lo cual se denomina *host*, la GPU es un coprocesador y se denomina *device*. Las ejecuciones de los procesos en la CPU están diseñadas para que se efectúen de manera secuencial, en cuánto las de la GPU en paralelo; las últimas realizándose en varios hilos de ejecución (*threads*). El nombre que recibe una función que ejecuta un proceso de forma paralela en la GPU se denomina *kernel*. *Host* y *device* poseen su propia memoria RAM, llamadas *host memory* y *device memory* respectivamente [7].

Los *threads* que posee una GPU se pueden agrupar de dos formas, una de ellas es por bloques (*thread block*) y la otra mediante grilla de bloques (*grid*). Todos los *threads* del mismo *thread block* poseen un acceso rápido a una memoria compartida y permite sincronizar las ejecuciones que se les asigna. Debido a la posibilidad de sincronización de los mismos, se evita el riesgo de que varios *threads* accedan de manera simultánea al mismo lugar de memoria. Una *grid* es un conjunto de *thread block*, en donde las instrucciones del *kernel* son paralelizadas, esto permite agrupar más *threads* de lo que está permitido por bloque. Puesto que la ejecución del proceso en los *thread blocks* de una *grid* pueden ser ejecutados en tiempos distintos, es necesario realizar un sincronización entre los mismos para que su comunicación sea segura y no haya conflictos [25]. La Figura (3.1) muestra el concepto de *grid* y *thread block*.

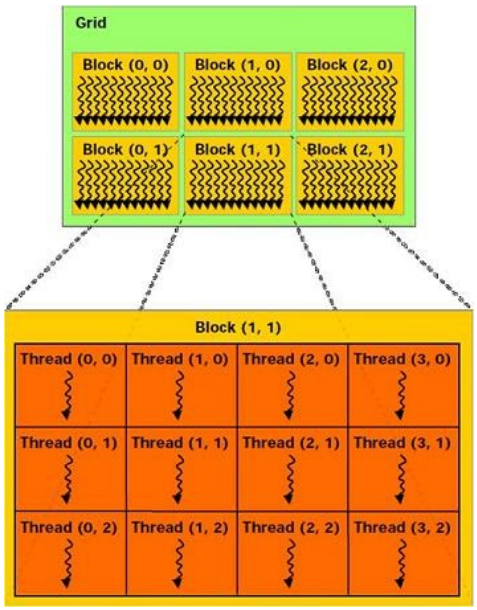


Figura 3.1: *Thread blocks* organizados en una *grid* [7].

En el proyecto se utilizaron dos PC, cada una con distinta CPU y GPU. En la Tabla (3.1) se encuentran las características de cada componente. La primera PC contaba con la CPU Intel Core i7-3770 y la GPU NVIDIA GeForce GTX 760. La segunda PC contaba con la CPU Intel Core i7-4770 y la GPU NVIDIA GeForce GTX 970.

Intel Core	i7-3770	i7-4770
Núcleos	4	4
Threads	8	8
Frecuencia de base [GHz]	3,40	3,40
Tamaño de memoria caché [MB]	8	8
Tipo de Memoria	DDR3-1333/1600	DDR3-1333/1600
Tamaño de Memoria [GB]	32	32
Ancho de banda de memoria [GB/s]	25,6	25,6

NVIDIA GeForce	GTX 760	GTX 970
Núcleos CUDA	1152	1664
Frecuencia de reloj normal [MHz]	980	1050
Tipo de Memoria	GDDR5 a 6 GHz	GDDR5 a 7 GHz
Configuracion de memoria estándar [MB]	2048	4096
Interfaz de Memoria [bits]	256	256
Ancho de banda de memoria [GB/s]	192.2	224.0

Tabla 3.1: Especificaciones técnicas de las CPU y GPU utilizadas [26][27][28][29].

3.2. Arquitectura de la memoria de una GPU

Según la arquitectura que posea un procesador, con su respectiva jerarquía en memoria, accesos y latencias, se planifica cómo llevar a cabo la implementación de un código, por ello es de importancia conocer dichas características.

Anteriormente se mencionó que el *host* y *device* poseen su propia memoria. La transferencia de datos de una memoria a otra tiene una muy alta latencia en cualquiera de los dos sentidos; por lo que al diseñar un código, hay que minimizar la transferencia de datos entre los dos tipos de memoria, y así obtener el menor tiempo de ejecución de los procesos.

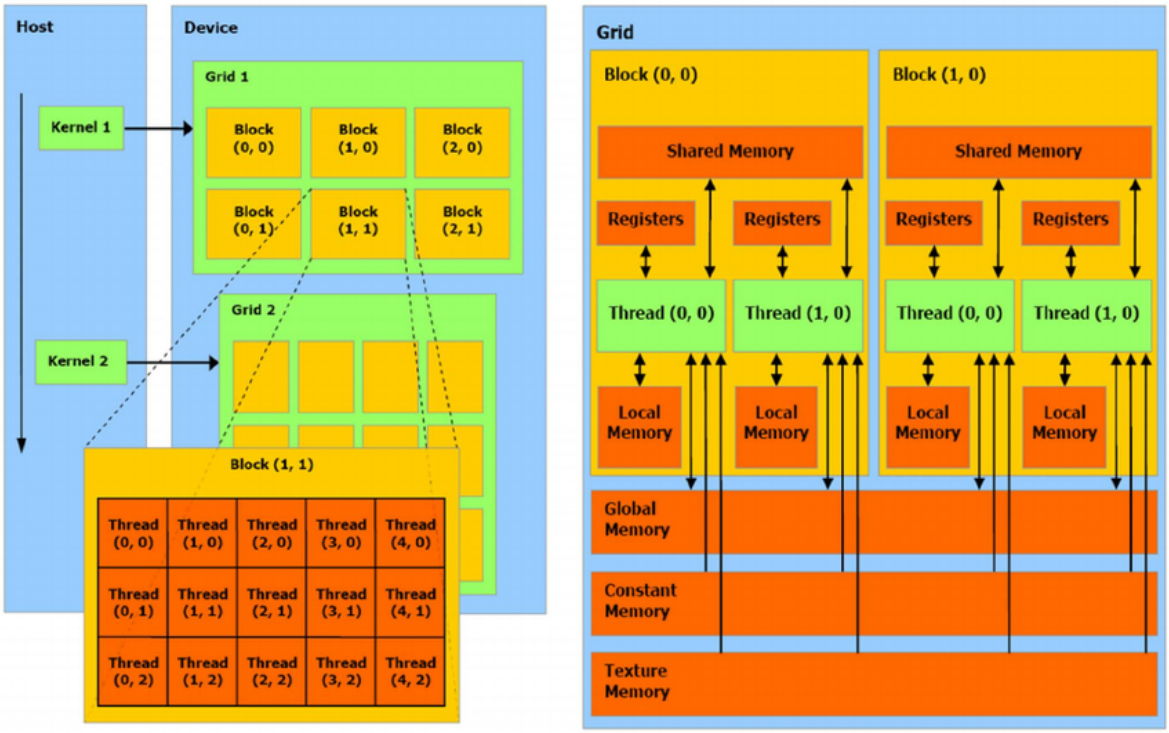


Figura 3.2: Esquematización de la arquitectura de CUDA. Izquierda: lanzamiento de un *kernel* desde el *host*. Derecha: jerarquía de memoria. [30].

Cuando el *host* efectúa su rutina de ejecución y se encuentra con un lanzamiento de *kernel*, éste será llevado a cabo en múltiples *threads* del *device*. Una representación de ello se muestra en la Figura (3.2).

Los procesos en el *device* tienen almacenados los datos según una jerarquía de memoria, con su respectiva limitación de acceso que se observa en la Figura (3.2). Los *threads* pueden acceder a datos de diversas memorias en distintos procesos. Estas memorias son las siguientes:

- *register memory* es visible para un único *thread*
- *local memory* tiene las mismas características que *register memory* pero con una performance menor.
- *shared memory* es visible por todos los *threads* de un mismo bloque, posee una baja latencia en su acceso.
- *global memory* es visible por todos los *threads* de la grilla y también por el *host*, posee una alta latencia de acceso.

  Las siguientes memorias poseen una alta latencia de acceso son asignadas y son aignadas para usos específicos y generalmente se almacenan en *caché*:

- *constant memory* es visible por todos los *threads* de la grilla siendo únicamente

de lectura. El uso de ésta memoria puede reducir el ancho de banda de memoria requerido en comparación con la global memory

- *texture memory* es otra memoria en la que sólo se puede leer y tienen acceso todos los *threads* de la grilla. Al realizar lecturas de *threads* ó *threadblocks* adyacentes su performance en comparación con *global memory* es mayor.

3.3. Programación en CUDA C

La programación de las GPU se lleva a cabo mediante el lenguaje CUDA C, el cual es una extención del lenguaje C. Debido su familiaridad y uso extendido, por lo que el lenguaje se denomina CUDA C. La versión 10.1 de CUDA fue la utilizada para el desarrollo del presente trabajo. La realización de procesos en paralelos es ejecutada mediante funciones llamadas *kernel* y son del tipo *void*.

Existen tres tipos de funciones que se pueden efectuar en CUDA C y son:

- **host** función clásica de C que se ejecuta en la CPU, siendo invocable únicamente por funciones que se realicen en la CPU.
- **global** es una función *kernel* invocada desde la CPU para ejecutarse en la GPU.
- **device** es una función que se ejecuta en la GPU y únicamente puede ser llamada desde un *kernel*.

En el presente trabajo sólo se utilizaran funciones de tipo **host** y **global**, pudiéndose realizar en un futuro el *profiling* mediante el uso de las funciones tipo **device**.

Otra particularidad que se presenta en la programación es el manejo de la memoria por parte del *host* como del *device*, siendo abordado posteriormente.

3.3.1. Programación de un *kernel*

Para visualizar las diferencias de programación de una función CUDA C (*kernel*), con una típica función de C, se muestra a continuación como ejemplo la programación para ambos lenguajes de la Ec. (2.7)

En primer lugar se detalla el código realizado en el lenguaje C, siendo la función :

.

```
void momentoDensity(scalar* rho, scalar* field, basicMesh* mesh);
```

.

donde el argumento **basicMesh* mesh** es un puntero a una estructura, la cual posee información del mallado que se realizó al dominio del problema a resolver, como por

ejemplo la cantidad de nodos que posee la malla (**nPoints**) y la cantidad de direcciones que posee el modelo en su espacio de velocidades **Q**. El tipo **scalar** corresponde a float o double, según opciones elegidas al momento de la compilación. El *array* **scalar* rho** es de dimensión *nPoints* y contiene los valores de ρ, por último **scalar* field** es un *array* que posee las q componentes de la función de distribución de poblaciones **f** para cada uno de los *nPoints* nodos de la malla.

```
#include <momentoDensity.h>
#include <stdio.h>

void momentoDensity(scalar* rho, scalar* field, basicMesh* mesh) {

        // Suma de todas las componentes

        for( uint i = 0 ; i < mesh->nPoints ; i++ ) {

                rho[i] = 0;

                for( uint j = 0 ; j < mesh->Q ; j++ ) {

                        rho[i] += field[ i*mesh->Q + j ];

                }

        }

}
```

La programación de la Ec. (2.7) implementada en CUDA C es con el *kernel*:

```
extern "C" __global__ void cudaMomentoDensity(
        cuscalar* field,cuscalar* rho, int np, int Q ) ;
```

en este caso se pasa de forma distinta los valores de la cantidad de nodos (np) y de la cantidad de velocidades del modelo $DdQq$ (Q). La distinción en los argumentos que se pasan en ambos códigos, es debido a la forma que CUDA C permite el manejo de las estructuras y también de las decisiones que se tomaron durante el desarrollo del código.

Para que un *kernel* compilado en formato PTX sea adquirido por el módulo PyCUDA de PYTHON con éxito, es que se indica como variable externa con el uso de extern "C".

.

```
#include <cudaMomentoDensity.h>
#include <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>

extern "C" __global__ void cudaMomentoDensity(cuscalar* field,
                                              cuscalar* rho,
                                              int np,
                                              int Q ) {

        int idx = threadIdx.x + blockIdx.x*blockDim.x;

        if( idx < np ) {

                int j= 0;

                cuscalar sum = 0;

                while ( j < Q ) {

                        sum += field[ idx*Q + j ];

                        j++;

                }

                rho[idx] = sum;

        }

}
```

.

Es de importancia conocer la función que cumplen las siguientes líneas: .

```
int idx = threadIdx.x + blockIdx.x*blockDim.x;
if( idx < np ) {
        ... (bloque de codigo)
}
```

.

donde las tareas que se hallen en if (idx $<$ np){...} se realizarán de forma paralela y es necesario identificar los *threads* en dónde se llevaran a cabo los procesos. El identificador de los *threads* es *idx* donde blockIdx.x indica la cantidad de *threads* por *block* y blockDim.x el numero de *block's*.

Resta ver cómo es el llamado de las funciones realizadas en el *main*, por lo que en C se observa: .

```
momentoDensity( rho, field_f, &mesh);
```

.

la cual no requiere de ninguna explicación, mientras que en CUDA C se tiene:

.

```
cudaMomentoDensity<<<ceil(mesh.nPoints/xgrid)+1,xgrid>>>(
                deviceField, deviceRho, cmesh.nPoints, cmesh.Q);
cudaDeviceSynchronize();
```

.

en donde $<<<, >>>$ indica la cantidad de *thread blocks* en que se realizará la tarea, como así también la cantidad de *threads* en cada *block*.

$$<<< \overbrace{ceil(mesh.nPoints/xgrid)+1}^{cantidad\ de\ threads\ por\ block} \ , \ \underbrace{xgrid}_{cantidad\ de\ thread\ block} >>>$$

3.3.2. Sincronización

Debido a que los procesos en la GPU son asignados de una forma no determinística, es necesario que exista un procedimiento para sincronizar los *threads*. Por ejemplo, un conflicto puede ser que el Thread idx = 1 quiera acceder a un lugar de la memoria al mismo tiempo que el Thread idx = 2. Otro caso es que el Thread idx = 1 escriba un valor en la *global memory* y se requiera que el Thread idx = 2 realice alguna operación con dicho valor. El método **cudaDeviceSynchronize()** fue el utilizado para resolver estos problemas, el cual sincroniza los *thread blocks* de una *grid*.

3.3.3. Utilización de la memoria de *host* y *device*

Las funciones que son realizadas en C permiten retornar alguna variable mientras que en CUDA C los *kernel* son del tipo *void*, debido a ello es necesario almacenar memoria en el *device* para una variable, y pasarla como argumento al *kernel*. Lo mencionado permite que el *kernel* modifique dicha variable y así obtener lo requerido.

La asignación de memoria en el *host* es la misma que se utiliza en C **malloc( )** : .

```
void *malloc(size_t size)
```

.

siendo *size_t* la memoria en bytes a reservar. Mientras que en el *device* se realiza por medio de **cudaMalloc( )** :

.

```
cudaMalloc(void **devPtr, size_t size);
```

.

donde *devPtr* es un puntero para asignar la memoria del *device* , *size_t* memoria en bytes a reservar. Se debe tener en cuenta que la memoria se reserva linealmente.

La transferencia de datos entre los dos tipos de memoria se efectua mediante la función **cudaMemcpy( )** : .

```
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind);
```

.

siendo *dst* un puntero con la dirección de destino de los datos, *src* un puntero con la dirección de origen de los datos, *count* es la cantidad de bytes a transferir y *kind* es el tipo de transferencia a realizar [31]. La Tabla (3.2) contiene los cuatro tipos posibles de transferencia.

TIPO DE TRANSFEREMCIA	SENTIDO DE TRANSFERENCIA
cudaMemcpyHostToHost	*host* ⟶ *host*
cudaMemcpyHostToDevice	*host* ⟶ *device*
cudaMemcpyDeviceToHost	*device* ⟶ *host*
cudaMemcpyDeviceToDevice	*device* ⟶ *device*

Tabla 3.2: Tipos de transferencias de datos en CUDA [32].

3.4. Programación en Python

La compilación de cada kernel (en CUDA C) en formato PTX permite que puedan ser incorporados en un programa de PYTHON mediante el módulo PYCUDA. El código de la siguiente página es el implementado para la Ec. (2.7).

La función *pycuda.driver.module_from_file()* nos permite importar el *kernel* compilado previamente en CUDA C, siendo **/PATH/** la dirección del directorio donde se encuentra el archivo PTX compilado. Las variables *deviceField* y *deviceRho* son las funciones de distribución de poblaciones **f** y la densidad ρ, respectivamente se deben haber inicializado las variables en el *host*, como por ejemplo: .

```
hostField = numpy.zeros( nPoints*meshQ, dtype=np.float32 )
```

.

y luego ser copiadas y asignadas en el *device* con la función *gpuarray.to_gpu()* : .

```
deviceField = gpuarray.to_gpu( hostField )
```

.

Además de los argumentos que tenía originalmente el *kernel*; hay que pasarle a la función obtenida por PYCUDA el número de *blocks* **block=(args.xgrid, 1, 1)** y la cantidad de *thread por block* **grid=(gs, 1, 1)** donde será ejecutada.

Por último la sincronización de los *threads* es proporcionada mediante el método *pycuda.driver.Context.synchronize()*.

```
import pycuda
from pycuda import gpuarray
import pycuda.autoinit

cudaMomentoDensity = pycuda.driver.module_from_file(
                        '/PATH/cudaMomentoDensity.ptx').get_function(
                        'cudaMomentoDensity')
cudaMomentoDensity(     deviceField,
                        deviceRho,
                        mesh.nPoints,
                        mesh.Q,
                        block=( args.xgrid, 1, 1 ),
                        grid=( gs, 1, 1) )
pycuda.driver.Context.synchronize()
```

3.5. Arquitectura del código numérico y compilación

El código numérico fue desarrollado en el repositorio de la página web GitHub de Microsoft Corporation, que provee un servicio de almacenamiento (*hosting*) de proyectos que utilizan la herramienta Git, la cual permite realizar un control de versiones. En la Sec. (3.6) se explica conceptos básicos para utilizar la herramienta Git, las funciones más relevantes y buenas prácticas que se adquirieron en el transcurso del presente trabajo. El proyecto se encuentra disponible para ser descargado en `https://github.com/efogliatto/LBCUDA_Test`, siendo la carpeta LBCUDA_Test la que contiene el proyecto. La versión 1.0 de la rama *master* es la que se poseía al momento de finalizar el presente trabajo.

Las subcarpetas (**en negrita**) y archivos (subrayados) que posee nuestro proyecto en su directorio principal son los siguientes:

- **build**: es el directorio de construcción, desde el cuál se compila el proyecto.
- **bin**: es el directorio en el cual se encuentran los ejecutables del proyecto.
- **include**: es el directorio en el cual se encuentran los *links* simbólicos a todos los *headers* de las funciones y *kernels* realizados.
- **lib**: es el directorio en el cual, una vez compilado el proyecto, se encuentran las bibliotecas realizadas en C y Cuda C.
- **ptx**: es el directorio en el cual, una vez compilado el proyecto, se encuentran compilados los *kernel* de Cuda C en formato ptx.

- **src**: es el directorio que posee los códigos fuente para los lenguajes C, CUDA C y PYTHON.
- **examples**: es el directorio el cual posee ejemplos de ejecución del código por medio de *scripts* de Bash[1].
- **validation_speed_up**: es un directorio que contiene dos tipos de *scripts* de Bash. Los del primer tipo al ser ejecutados, realizan la validación de los problemas tratados en el Cap. (4). El otro tipo de *scripts* realizan una toma de tiempo de cálculo de los mismos problemas, para obtener un índice (*speed_up*) que indique la mejora de tiempo de cálculo entre lenguajes utilizados.
- CMakeLists.txt: es el archivo de configuración de CMAKE, con la definición de las principales opciones de compilación y *linkeo*.

La herramienta de software CMAKE, versión 3.17.2, fue la utilizada para preparar la compilación del código desarrollado, en donde los compiladores utilizados son *gcc* y *nvcc*. El archivo CMakeLists.txt principal tiene que encontrarse en la carpeta principal del proyecto [33], y contiene todas las instrucciones de compilación que tiene el proyecto a trabajar. A continuación se detallan las instrucciones más relevantes.

Para indicar las carpetas de construcción (*build*), ejecutables (*bin*), bibliotecas (*lib*), *headers* de las funciones y *kernels* (*include*) y código fuente (*src*) se incorporan las siguientes instrucciones:

```
set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}/build)

set(EXECUTABLE_OUTPUT_PATH ${CMAKE_SOURCE_DIR}/bin)

set(LIBRARY_OUTPUT_PATH ${CMAKE_SOURCE_DIR}/lib)

include_directories(include) # Bring the headers

add_subdirectory (src)  # Directorio con fuentes
```

donde los comandos que empiezan con CMAKE_ son variables, que son la unidad básica de almacenamiento en el lenguaje CMake, y la lista completa de las mismas se encuentra en `https://cmake.org/cmake/help/v3.17/manual/cmake-variables.7.html`.

Para asignar las bibliotecas en donde el código será compilado, se incorporan las instrucciones mostradas a continuación, donde se colocaron algunas bibliotecas utilizadas a modo de ejemplo .

[1]GNU Bash o simplemente Bash es una terminal para ejecutar comandos de un sistema operativo de Unix.

```
# Link libraries

#Para el codigo realizado en C
set ( PROJECT_LINK_LIBS
liblatticemodel.so
liblatticemesh.so
...
)

#Para el codigo realizado en Cuda C
set ( CUDATEST_LINK_LIBS
cudatest.a
cudalatticemesh.a
...
)

link_directories( ${CMAKE_SOURCE_DIR}/lib )
```

La instrucción de asignar opciones de compilación en C y CUDA C respectivamente son las siguientes:

SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} (opciones)")

SET(CMAKE_CUDA_FLAGS "${CMAKE_CUDA_FLAGS} (opciones)")

donde es importante señalar que dependiendo de la arquitectura que posee la GPU, debe asignársele una distinta opción de compilación. A continuación su muestra una opción de compilación genérica, en donde xx se refiere a la arquitectura de la GPU.

SET(CMAKE_CUDA_FLAGS "${CMAKE_C_FLAGS}
. -gencode arch=compute_xx,code=sm_xx -rdc=true")

Otras de las instrucciones que se le asignó al compilador, es que tenga la opción de que el proyecto se compile únicamente en C ó, en su defecto C y CUDA C. A su vez como se trabajó con distintas GPUs, se implementó que el compilador detecte de forma automática la arquitectura que poseía la GPU de la PC, en donde se está realizando la compilación.

El siguiente archivo de configuración de CMAKE muestra la implementación de las instrucciones descriptas arriba:

```
SET( CMAKE_C_FLAGS  "${CMAKE_C_FLAGS} -Wall -lm -pedantic -funroll-loops" )

# Opcional: Desabilitar CUDA para poder compilar al menos
# una parte en dispositivos sin GPU

option(DISABLE_CUDA "Deshabilitar manualmente la compilacion en CUDA" OFF)

set(CUDAARCH "30" CACHE STRING "Arquitectura de GPU")

if(DISABLE_CUDA)

        # Asignar solo C como lenguaje de compilacion
        project(LBCUDA_Test LANGUAGES C)

else()
        # Asignar C y CUDA como lenguajes
        project(LBCUDA_Test LANGUAGES CUDA C)

        # Bibliotecas para compilar CUDA
        find_package( CUDA REQUIRED )

        # Opciones de compilador
        if(CUDAARCH STREQUAL "30")
                message("-- CUDA: Arquitectura 30")
                SET( CMAKE_CUDA_FLAGS  "${CMAKE_CUDA_FLAGS}
                                -gencode arch=compute_30,code=sm_30 -rdc=true")
        else()
                if(CUDAARCH STREQUAL "50")
                        message("-- CUDA: Arquitectura 50")
                        SET( CMAKE_CUDA_FLAGS  "${CMAKE_CUDA_FLAGS}
                                        -gencode arch=compute_50,code=sm_50 -rdc=true")
                else()
                        message("-- CUDA: Arquitectura no reconocida. Uso de 30")
                        SET( CMAKE_CUDA_FLAGS  "${CMAKE_CUDA_FLAGS}
                                        -gencode arch=compute_30,code=sm_30 -rdc=true")
                endif()
        endif()
endif()
```

Por último el código numérico se realizó para que sus variables estén en simple o doble precisión. Por lo cual en el código de C, la declaración de variables que tomarán la asignación de float ó double será **scalar**, mientras que en el código de CUDA C será **cuscalar**.

La forma de concretar lo mencionado se realizó en dos partes. La primera de ellas es que en el CMakeLists.txt principal se declararon las variables SP, DP, SP_CUDA y DP_CUDA, correspondiendo a simple precisión, doble precisión en C y CUDA C respectivamente. Por lo que dependiendo de que variable es la adoptada en el compilador, será la que se utilice luego en el código. El archivo de configuración de CMAKE que ejecuta lo propuesto es el siguiente:

```
# Optional: precision
set(PRECISION "simple" CACHE STRING "Floating-point precision")

if(PRECISION STREQUAL "double")
        message("-- C   : Double precision")
        add_definitions(-DDP)

        elseif(PRECISION STREQUAL "simple")
                add_definitions(-DSP)
                message("-- C   : Simple precision")
        else()
                message( FATAL_ERROR "Precision not supported" )
endif()

# Optional: CUDA precision
set(CUDA_PRECISION "simple" CACHE STRING "CUDA Floating-point precision")

if(CUDA_PRECISION STREQUAL "simple")
        message("-- CUDA: Simple precision")
        add_definitions(-DSP_CUDA)

        elseif(CUDA_PRECISION STREQUAL "double")
                add_definitions(-DDP_CUDA)
                message("-- CUDA: Double precision")
        else()
                message( FATAL_ERROR "CUDA Precision not supported" )
endif()
```

La segunda parte para asignar un tipo de variable a **scalar** y **cuscalar** es el archivo *header* que aparece debajo. En este se busca las variables declaradas en el compilador (SP, DP, SP_CUDA ó DP_CUDA) y en base a estas, define el tipo de variable para **scalar** y **cuscalar**. El *header*, cuyo nombre de archivo es *dataTypes.h* y se encuentra en el directorio *include*, debe ser incluido en cada función o *kernel* que posea una variable de tipo **scalar** o **cuscalar**.

.

```
        #ifndef DATATYPES_H
        #define DATATYPES_H
                #ifdef DP
                        typedef double scalar;
                #elif SP
                        typedef float scalar;
                #endif

                #ifdef DP_CUDA
                        typedef double cuscalar;
                #elif SP_CUDA
                        typedef float cuscalar;
                #endif
        #endif // DATATYPES_H
```

Anteriormente se detalló cómo agregar la carpeta con el código fuente (*src*), este a su vez posee otras subcarpetas, divididas por el tipo de lenguaje que posee el código. Para que CMAKE incorpore estas carpetas con código fuente, se crean archivos CMakeLists.txt en cada subcarpeta y se agrega la ruta de compilación a la misma mediante el método **add_subdirectory()**. Lo descripto se realiza con el siguiente archivo de configuración de CMAKE :

.

```
# Bibliotecas y aplicaciones en C
add_subdirectory (C)

# Bibliotecas y aplicaciones en CUDA
if(DISABLE_CUDA MATCHES OFF)
        add_subdirectory (CUDA)
endif()
```

La Figura (3.3) muestra la arquitectura principal del proyecto a partir del directorio ***src***. Se encuentra desglosada las subcarpetas del lenguaje C, ya que la carpeta de CUDA C es similar. Para PYTHON sólo se implementó el código con uno de los *kernels* (compilados en PTX), por lo que no se posee una estructura como la del resto de los lenguajes.

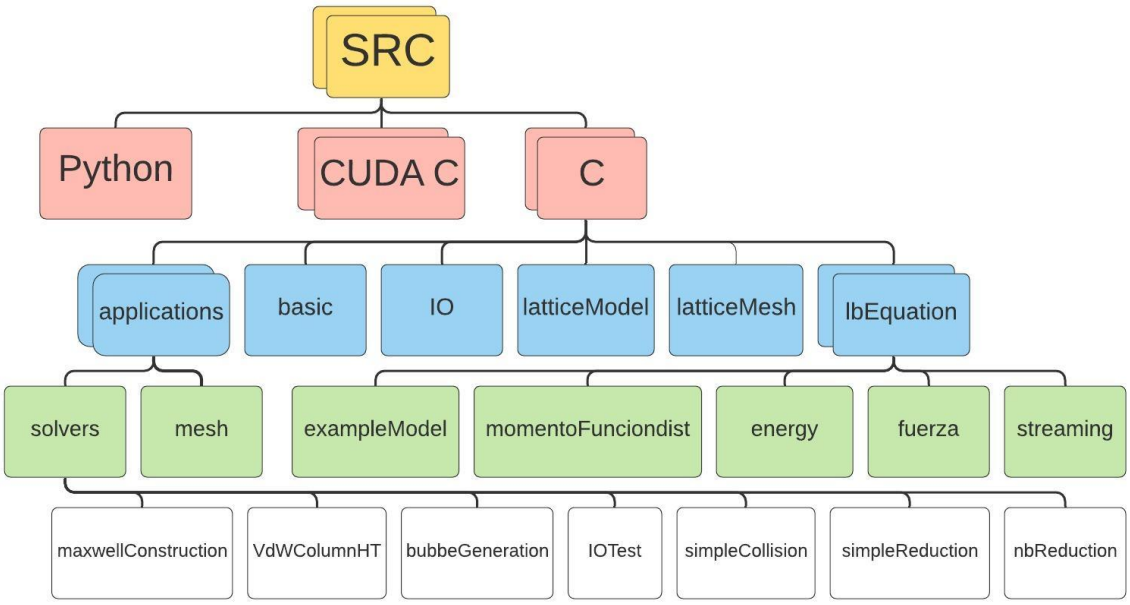


Figura 3.3: Esquema de la arquitectura del proyecto a partir del directorio SRC, el cual posee todos los códigos fuente. La carpeta de CUDA C presenta la misma estructura mostrada que la de C.

Cada una de las carpetas finales que se encuentran detalladas poseen uno o varios archivos que son funciones o *kernels* con sus respectivos *headers* y cada uno de los *headers* tiene un *link* simbólico en el directorio *include*.

Para agregar las funciones o *kernels* a las bibliotecas que se configuraron en el CMakeLists.text principal, se debe indicar una instrucción **add_library()**. Primero se ejemplifica en el código de C para la biblioteca *lbequation.so* con algunas de las funciones que posee la misma. En el directorio de lbEquation, además de contar con sus subdirectorios, posee el siguiente archivo de configuración de CMake:

```
#------------------ Generacion de bibliotecas ------------------#

                # Esquemas LB
                add_library(lbequation SHARED
                exampleModel/exampleDensity.c
                ...

                momentoFunciondist/momentoDensity.c

                fuerza/fuerzaPresionEOS.c
                ...

                streaming/lbstreaming.c
                ...

                energy/energyTemp.c
                ...

                )
```

En el caso del código de Cuda C, además de lo realizado, se debe agregar la instrucción de compilación en el formato ptx de cada uno de los *kernels* desarrollados. El archivo de configuración de CMake de la siguiente página ejemplifica la forma de obtener lo requerido.

El directorio *applications* posee archivos que contienen la definición de *main* y son los ejecutables. Es necesario remarcar dos aspectos. Por un lado la estructuración de la memoria para resolver los problemas mediante el método descripto en la Sec. (2.4). Por otro lado la instrucción que debe dársele al compilador para que efectivamente el archivo sea compilado como un ejecutable.

```
# Biblioteca de pruebas para CUDA C/CXX
add_library(cudalbequation STATIC

cudaExampleModel/cudaExampleDensity.cu
...

cudaMomentoFunciondist/cudaMomentoDensity.cu
...

...

)

set_target_properties( cudalbequation
                PROPERTIES CUDA_SEPARABLE_COMPILATION ON)

# Compilacion en PTX para uso de PyCUDA
add_library(cudalbequationPTX OBJECT

cudaExampleModel/cudaExampleDensity.cu
...

cudaMomentoFunciondist/cudaMomentoDensity.cu
...

...
)

set_property(TARGET cudalbequationPTX PROPERTY CUDA_PTX_COMPILATION ON)

install(TARGETS cudalbequationPTX
                OBJECTS DESTINATION ${CMAKE_SOURCE_DIR}/ptx )
```

La estructura de la memoria utilizada, es reservada en matrices o vectores para todos los elementos de la malla con los parámetros que se detallan a continuación.

- Función distribución de poblaciones **f** y **q**. (Tamaño N x q)
- un *array* Swap para realizar el proceso de *streaming*. (Tamaño N x q)
- información de elementos vecinos de un nodo de la malla (Tamaño N x q)
- Densidad ρ. (Tamaño N)
- Temperatura T. (Tamaño N)

- Velocidad **U**. (Tamaño N x 3)
- Fuerza de interacción entre elementos de malla $\mathbf{F_{int}}$. (Tamaño N x 3)
- Fuerza de total actuante en los nodos $\mathbf{F_{tot}}$. (Tamaño N x 3)

En éste caso N es la cantidad de elementos de la malla y q la dimensión de la velocidad de grilla del modelo.

Cabe destacar que la matriz que posee la información de los nodos vecinos, sigue una asignación utilizada según cómo está dispuesta la velocidad de grilla **e**, como indica la Figura (1.2). En nuestro caso, la información se encuentra almacenada para el nodo i de la siguiente forma:

$$vecino_i = \begin{bmatrix} v_0 & v_1 & v_2 & v_3 & v_4 & v_6 & v_7 & v_8 \end{bmatrix}$$

donde la nomeclatura de v_α indica que es el vecino que se encuentra en la dirección de α. En el caso presentado $v_0 = i$. Ésta disposición marca la forma en que leerán los registros de memoria, pudiendo no estar optimizada totalmente.

Para finalizar se utiliza la instrucción **add_exacutable()** y **target_link_libraries()** para realizar un archivo ejecutable mediante CMake. A modo de ejemplo, se muestra el archivo CMakeLists.txt del ejecutable maxwellConstruction que se encuentra en el directorio con su mismo nombre.

.

```
#------------------ Construccion de Maxwell------------------#
# Construccion de Maxwell

add_executable(maxwellConstruction
                        "maxwellConstruction.c" "writeDebug.c")
target_link_libraries(maxwellConstruction
                        ${PROJECT_LINK_LIBS} ${PROJECT_LINK_LIBS})
```

3.6. Control de versiones utilizando GIT

Git es un software que sirve para controlar versiones de archivos que se encuentran en un repositorio. El mismo fue desarrollado por Linus Torvalds cuando él empezo a desarrollar el *kernel* de *Linux*. El proyecto de éste trabajo se desarrollo con el sistema operativo *Debian 10* y se encuentra en el repositorio de la página web de GITHUB. El proyecto puede ser descargado mediante las siguientes líneas de comando en la terminal de *Linux* que crean un repositorio local del mismo .

```
$ sudo apt-get install git #Para instalar Git

$ git clone https://github.com/efogliatto/LBCUDA_Test #Iniciar descarga
```

3.6.1. Conceptos básicos

Para entender el funcionamiento de GIT se definen algunos conceptos básicos [34].

El directorio de trabajo en que se desarrolla el proyecto se llama *working directory* y el estado de sus archivos pueden ser nuevo, modificado o eliminado. El seguimiento (*tracked*) de un archivo permite almacenar toda la información del mismo, y se realiza al añadirlo (*add*) al espacio llamado *stage index*. Una vez que han sido añadido al *stage index* los archivos con su estado deseable, es que se realiza un *commit* para guardar los cambios realizados en el espacio denominado *repository*.

Un *commit* posee una identificación y se le asigna un comentario que debe reflejar las diferencias desarrolladas con respecto al *working directory* del *commit* anterior. Una rama (*branch*) es una línea de trabajo, en donde se desarrolla el proyecto y se almacenan todos los cambios que permiten determinar alguna funcionalidad en particular. Se pueden tener múltiples ramas para trabajar un proyecto y obtener un mejor desarrollo del mismo. Cuando es conveniente, una rama que terminó su desarrollo puede ser fusionada en otra, éste proceso se llama *merge*.

GIT fue pensado para hacer desarrollo entre colaboradores. Lo más usual es contar con un repositorio remoto principal, en donde se encuentra el último estado del proyecto a trabajar y eventualmente todas las ramas que se esten desarrollando. Cada uno de los colaboradores posee su propio repositorio, dependiendo de si es el propio o de otro colaborador recibe el nombre de local y remoto. Debido al trabajo continuo de los colaboradores, el repositorio remoto puede ir variando. El proceso de subir los cambios de alguna rama del repositorio local al repositorio remoto se llama *push* y en el sentido opuesto se denomina *pull*[2].

[2] *pull* fusiona automáticamente el estado del repositorio remoto en el local, esto puedo ocasionar conflictos y por ello es que se suele utilizar *fetch*. Para mayor información remitirse al manual de usuario `https://git-scm.com/docs/user-manual`.

3.6.2. Principales comandos

En el transcurso del desarrollo del presente proyecto se utilizaron algunos de los comandos de GIT. A continuación se detallan los más utilizados[3]:

- **git init**: inicializa un repositorio local.
- **git clone URL**: copia un repositorio que se encuentra en la dirección URL.
- **git add**: añade los archivos que se le indiquen al *stage index*.
- **git status**: indica el estado actual del *repository*, la rama en la que se encuentra y los archivos que han sido creados en el *working directory* y que no se ha realizado un *tracked*.
- **git commit -a**: se realiza un *commit*, donde el *flag a* realiza una preparación automática de todos los archivos trackeados, ahorrando el paso de **git add** [34].
- **git log**: se muestra los comentarios de todos los *commits* realizados con su nombre indicatorio. Si se coloca **-n** como opción, se mostrarán los últimos n *commits*.
- **git commit - -amend**: se utiliza cuando se realizo mal el comentario de un *commit* o faltó agregar algún archivo.
- **git push**: se añade al repositorio remoto todos los *commits* realizados de alguna rama.
- **git pull**: se adquiere del repositorio remoto, los *commits* que no se poseen en el local de alguna rama.
- **git checkout name**: el repositorio local vuelve al estado del nombre identifiatorio sin perder lo trabajado y permite cambiar de rama.
- **git branch name_branch**: si se desea modificar desde un cierto *commit* se debe crear una nueva rama y a partir de ahí seguir el desarrollo, luego de ello se debe realizar **git checkout name_branch**.
- **git diff**: posee varias opciones y nos muestra diferencias. Ellas pueden ser de un archivo de la rama actual, entre dos ramas, entre un archivo que se encuentra en dos ramas, entre otros.
- **git merge**: se utiliza para fusionar dos ramas. Si no hay conflictos entre las dos ramas que se fusionan se realiza un *fast foward* y no se debe realizar nada. Si hay conflictos y no se realiza el *merge* de forma automática, se debe resolver los conflictos[3].

[3]Para más información remitirse al manual de usuario `https://git-scm.com/docs/user-manual` o en su defecto directamente al libro `https://git-scm.com/book/en/v2`.

3.6.3. Buenas prácticas

En el transcurso del proyecto se adquirieron buenas prácticas de desarrollo de código mediante GIT, dónde algunas de ellas son las que se desarrollan a continuación.

Commit: al momento de realizar un commit, es recomendable no hacerlo sobre muchos archivos juntos, sino sobre algunos que reflejen un cambio en la misma línea de trabajo. El comentario que lleva el *commit* debe ser corto y enfocado a un desarrollo o cambio puntual. Antes de hacer **push** al repositorio remoto verificar que todos los archivos que se desea esten añadidos, caso contrario se los agrega y para no generar otro *commit* es que se utiliza **git commit - -amend**.

Branch: se utiliza para facilitar el seguimiento del grado de avance del proyecto. Es recomendable trabajar con tres (3) niveles de ramas, el primero es de la rama *master* que posee la última versión estable[4], la segunda es la rama *develop* donde se harán los desarrollos (que compila y contiene algunas funcionalidades incorporadas funcionado) y la tercera capa posee una o varias ramas tipo *feature* que se enfocan en desarrollar algo específico. La idea es que cuando se tengan terminadas las versiones del *feature*, se realice en *merge* a la rama *develop* y una vez que ésta posea todos los *feature* hacer un MERGE a la rama *master*. Lo que se explico queda ilustrado mediante la Figura (3.4).

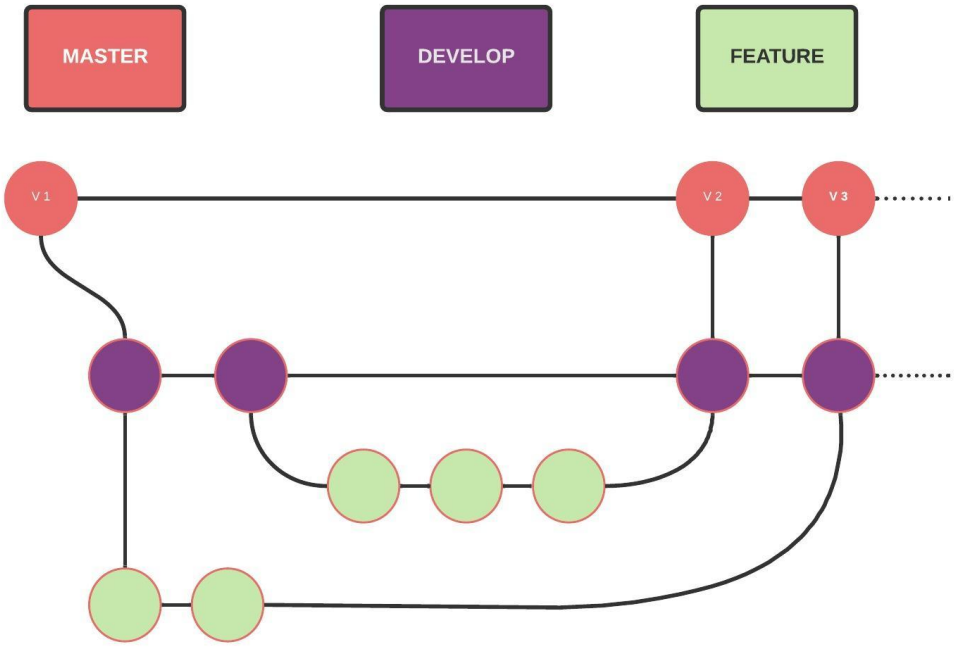


Figura 3.4: Ejemplo de un diagrama de la gestión y control de versiones del proyecto empleando ramas utilizado en el presente trabajo.

[4]En el presente trabajo se considera una versión como estable si la misma fue validada.

Capítulo 4

Validación de la herramienta numérica

En el presente capítulo se realiza la descripción de los problemas con transferencia de calor para fluidos multifásicos con cambio de fase llevados a cabo para validar los códigos numéricos desarrollados, siendo éstos la *Construcción de Maxwell*, la *Estratificación de un fluido Van der Waals con temperatura no uniforme* y la *Generación de burbujas en una superficie horizontal calefaccionada.*

4.1. Construcción de Maxwell (2D)

En esta sección se explica el procedimiento por el que a partir de una EOS, se puede obtener las densidades de coexistencia de un fluido para un dado valor de presión y temperatura. Dicho procedimiento se llama Construcción de Maxwell, donde necesariamente la EOS con la que se modela el fluido debe permitir la coexistencia de densidades para sus variables de estado. Para la resolución de éste problema solo se utiliza la ecuación hidrodinámica del LBM.

En el desarrollo del Cap. (2) se adoptó como EOS del presente trabajo, la ecuación de VdW Ec. (4.1) que describe el comportamiento de un gas real y permite la coexistencia de diferentes densidades para igual presión y temperatura:

$$p = \frac{RT}{V_m - B} - A\left(\frac{1}{V_m}\right)^2 \tag{4.1}$$

La Ec. (4.1) puede ser representada gráficamente en un diagrama $P - V_m$. A modo de ejemplo se presenta el diagrama para el caso del dióxido de carbono (CO_2) a tres temperaturas distintas ver Figura (4.1). La temperatura de interés es la de $T = 270\,K$, donde se vislumbra que en $p = 44,08\,atm$ la gráfica se intersecta en tres valores de V_m, siendo dos de ellos estables; por lo que se observa que hay dos volúmenes molares de coexistencia, indicando las fases líquido y gaseosa [16]. Además se forman las áreas **A1** y **A2** delimitadas por la curva $P - V_m$ y la línea de presión constante a $p = 44,08\,atm$.

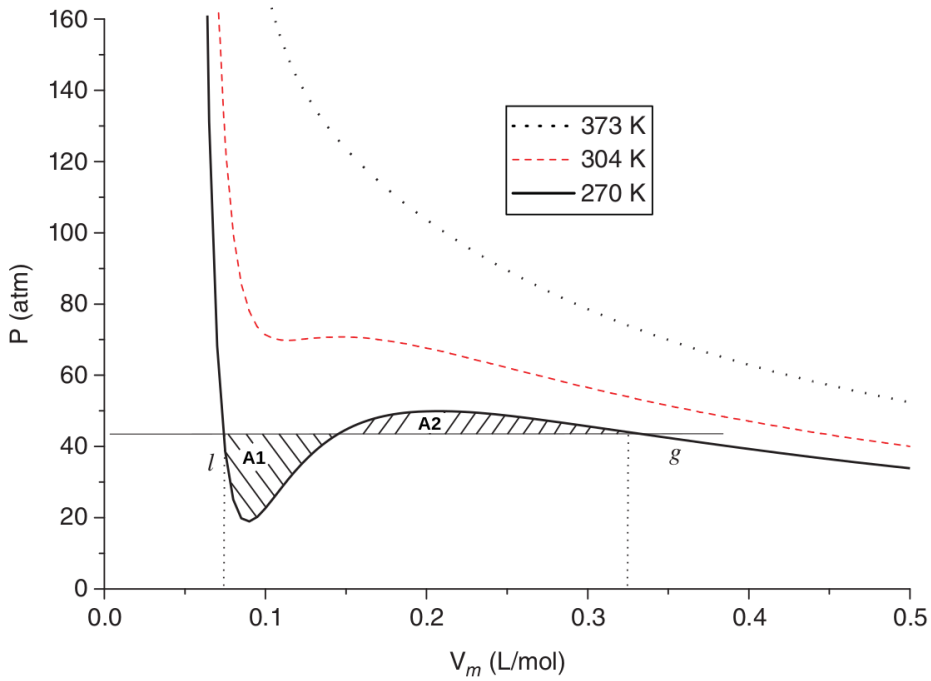


Figura 4.1: Diagrama $P - V_m$ de la EOS de VdW del CO_2 con las constantes $A = 3,592$ y $B = 0,04267$, representando para $T = 270\,K$ los volúmenes molares del líquido y gas. **A1** y **A2** son el área encerrada entre la curva $P - V_m$ a $T = 270\,K$ y la línea de presión constante a $p = 44,08\,atm$ [16].

La construcción de Maxwell, también llamada regla de igualdad de áreas, indicada en la Ec. (4.2), es un procedimiento analítico para encontrar las densidades de coexistencia del líquido y gas. Al realizar la integral propuesta surge que las áreas **A1** y **A2** de la Figura (4.1) deben ser iguales.

$$\int_{V_{m,l}}^{V_{m,g}} P\, dV_m = p_0(V_{m,l} - V_{m,g}) \tag{4.2}$$

En donde P es la presión de la EOS y p_0 es una presión constante.

La Ec. (4.1) puede reescribirse como:

$$p_{EOS} = \frac{\rho RT}{1 - \rho b} - a\rho^2 \tag{4.3}$$

donde $\rho = \frac{1}{v}$, siendo $v = \frac{V_m}{M}$ el volumen másico. De esta forma reescribiendo la integral de la Ec. (4.2) usando la presión de Ec. (4.3), pueden obtenerse los valores de coexistencia ρ_l y ρ_g para las fases líquida y gaseosa, respectivamente.

La coexistencia de fases tiene lugar a temperaturas inferiores a una temperatura crítica (T_c). En el ejemplo mostrado de la Figura (4.1) $T_c = 304\,K$. Analíticamente T_c surge de aplicar el criterio de la primera y segunda derivada a la Ec. (2.7) como se indica en Ec. (4.4) y se deben conocer los parámetros a y b.

$$\frac{\partial\, p}{\partial\, V_m} = 0 \qquad \frac{\partial^2\, p}{\partial\, {V_m}^2} = 0 \tag{4.4}$$

Realizando adecuadamente la adimensionalización de la Ec. (4.3) se puede graficar una curva de coexistencia $T_r - \rho_r$, donde $T_r = \frac{T}{T_c}$, $\rho_r = \frac{\rho}{\rho_c}$ y T_c es calculado a partir de los parámetros a y b asignados al fluido de estudio. Es de importancia destacar que las curvas de coexistencia dependen de la EOS que se esté utilizando para describir el comportamiento del fluido. En el caso de estudio analizado, la EOS es de VdW y la curva de coexistencia es universal, independientemente de los parámetros a y b, debido a que se encuentra normalizada por unidades críticas.

Para un fluido con una EOS de VdW con sus respectivos parámetros a y b, se puede calcular los valores de T_c y ρ_c. Se inicializa aleatoriamente la densidad del fluido mencionado alrededor de ρ_c, en una cavidad bidimensional que se encuentra a una dada temperatura. A consecuencia de la evolución temporal se producirá la separación de las fases del mismo, produciéndose una coexistencia de fases. Por medio de la obtención de las densidades de coexistencia para distintas temperaturas iniciales, se puede reconstruir la curva de coexistencia analítica. De esta forma, el problema de construcción de Maxwell puede emplearse para validar la implementación de las ecuaciones LBM hidrodinámicas.

En este problema, se valida la curva de coexistencia $T_r - \rho_r$ calculada analíticamente para un fluido con una EOS de VdW, como la observada en la Figura (4.2).

4.1.1. Validación

La validación de este problema se hizo utilizando los parámetros $a = 0,5$ y $b = 4,0$; para un tamaño de malla de 201 x 201 nodos y T_r variando con un paso de $0,025$ en el rango de $[0,6 - 0,975]$. El valor de los parámetros utilizados de LBM son los siguientes:

$$diag(\mathbf{\Lambda}) = \begin{bmatrix} 1{,}0 & 0{,}8 & 1{,}1 & 1{,}0 & 1{,}1 & 1{,}0 & 1{,}1 & 0{,}8 & 0{,}8 \end{bmatrix}$$
$$G = -1{,}0 \quad c = 1{,}0 \quad \sigma = 0{,}125 \quad a = 0{,}5 \quad b = 4{,}0$$
$$\mathbf{g} = (0{,}0 \quad 0{,}0 \quad 0{,}0) \qquad \rho_c = \frac{1}{12} \qquad T_c = 0{,}037037037$$

La Figura(4.2) muestra la validación del código realizado en C para simple precisión en una CPU Intel Core i7-3770; con distintos parámetros σ del modelo MRT, donde se observa que el valor de $\sigma = 0,125$ es el que mejor ajusta a la curva de coexistencia como se discute en [8]. Por otro lado, en la Figura (4.3) se muestran los resultados obtenidos con el código realizado en CUDA C, y ejecutado en la GPU correspondiente

Los resultados, que se muestran en las Figuras (4.2) y (4.3), se realizaron en 50000

pasos de tiempo en cada uno de los valores de T_r. Puede observarse que los valores obtenidos de las densidades de coexistencia de fases entre los códigos de C y CUDA C resultan similares.

Al comparar los resultados luego de la validación de las densidades de fase entre C y CUDA C para doble precisión, se obtuvo que poseen el mismo valor numérico.

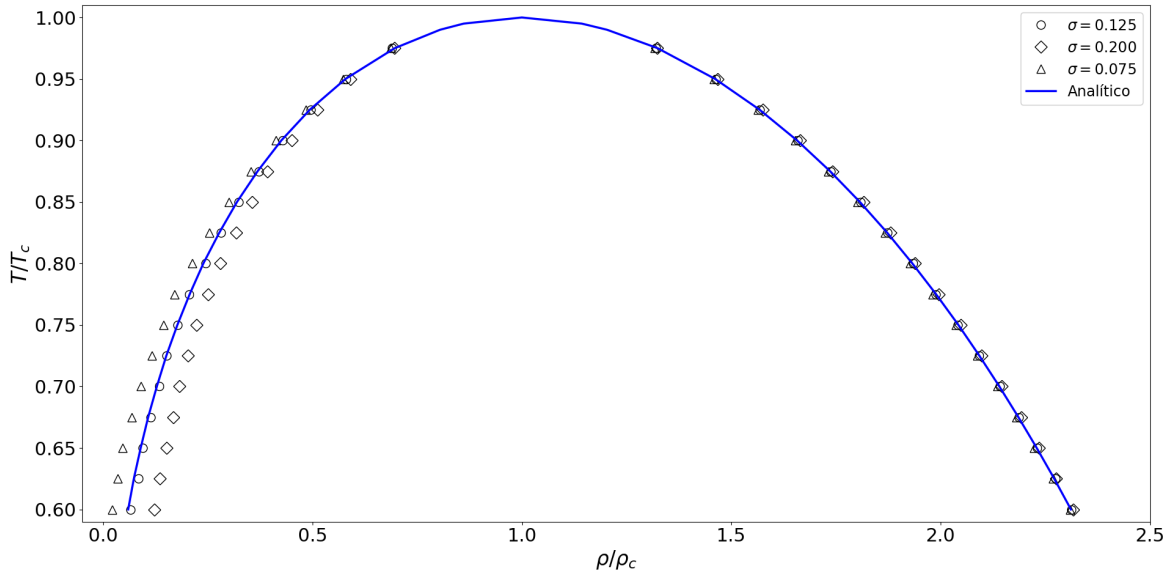


Figura 4.2: Curva de coexistencia de fases para un fluido de VdW con los parámetros $a = 0,5$ y $b = 4,0$, obtenida en simple precisión en la CPU Intel Core i7-3770 en el código desarrollado en C. Los puntos corresponden a valores del parámetro libre del modelo MRT de $\sigma = 0{,}075[\triangle]$ $\sigma = 0{,}125[\bigcirc]$ y $\sigma = 0{,}200[\diamondsuit]$

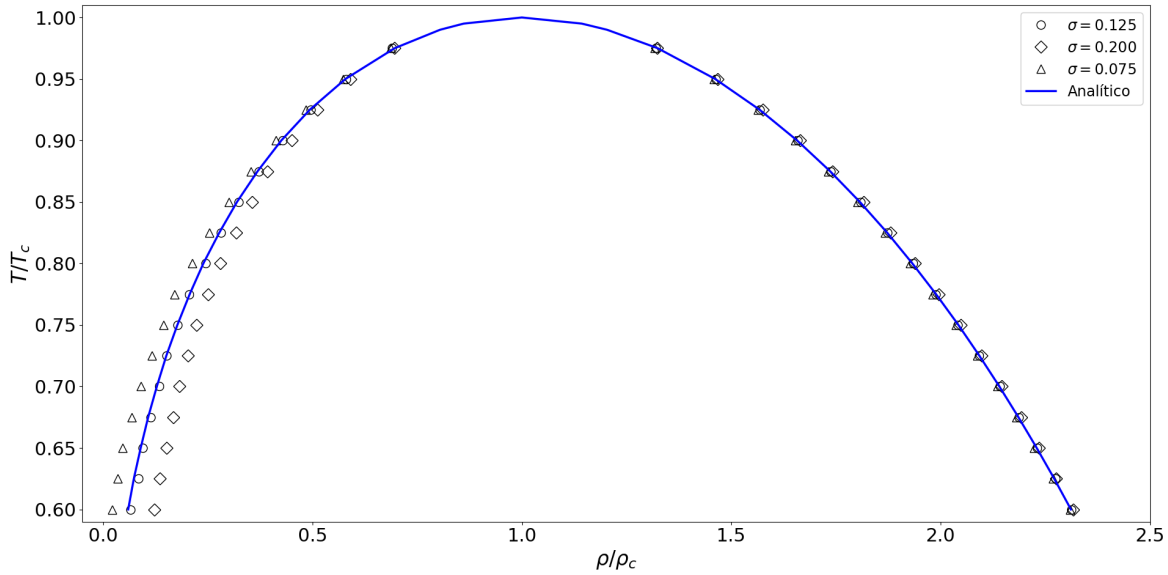


Figura 4.3: Curva de coexistencia de fases para un fluido de VdW con los parámetros $a = 0,5$ y $b = 4,0$, obtenida en simple precisión en la GPU NVIDIA GeForce GTX 760 en el código desarrollado en CUDA C. Los puntos corresponden a valores del parámetro libre del modelo MRT de $\sigma = 0{,}075[\triangle]$ $\sigma = 0{,}125[\bigcirc]$ y $\sigma = 0{,}200[\diamondsuit]$

4.1.2. Comparación de precisiones

La Figura (4.4) muestra los resultados de las densidades de coexistencia de fases para simple y doble precisión, no siendo apreciable la diferencia.

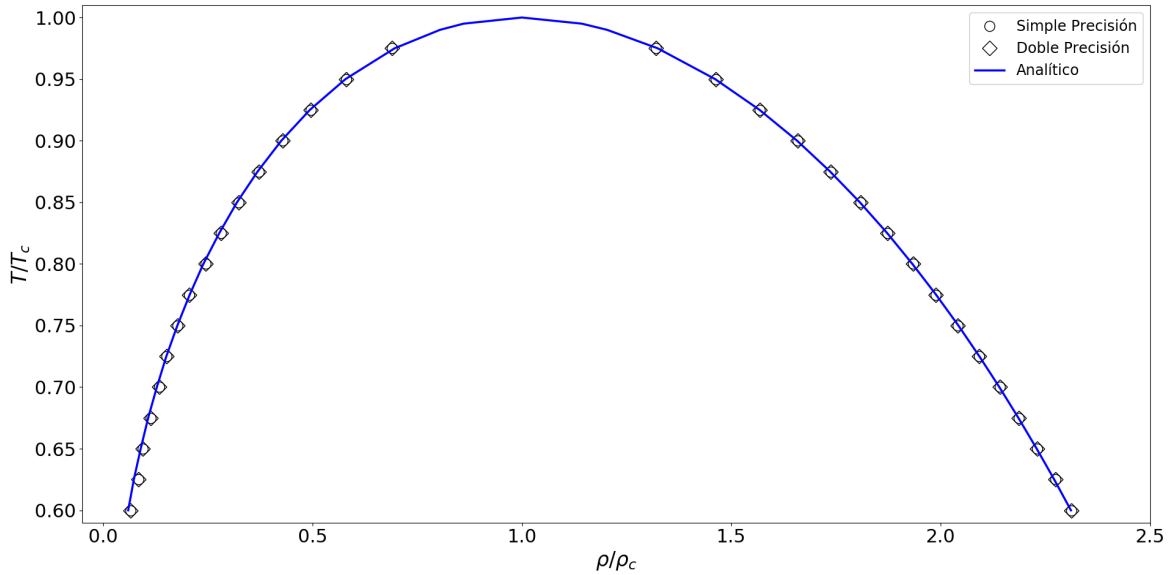


Figura 4.4: Curva de coexistencia de fases para un fluido de VdW con los parámetros: $a = 0,5$ y $b = 4,0$. Los puntos corresponden a los valores obtenidos en simple precisión [◯] y doble precisión [♢] en la GPU NVIDIA GeForce GTX 760 en el código desarrollado en C.

El error en la aproximación a la solución analítica se mide como la distancia entre los vectores de densidad de fase obtenido y el analítico. La distancia se calculó por medio de la Norma Euclídea, siendo calculada la distancia entre dos vectores $\boldsymbol{A}$ y $\boldsymbol{B}$ con i elementos como indica la Ec. (4.5):

$$dist(\mathbf{A}, \mathbf{B}) = \sqrt{\sum_i (a_i - b_i)^2} \tag{4.5}$$

La mayor diferencia porcentual hallada para la distancia en la fase gaseosa, es que en simple precisión es 0,0034 % mayor que en doble precisión. Para la fase líquida se obtuvo que en doble precisión es 0,0491 % mayor la distancia que en simple precisión.

4.1.3. Speed Up

En la presente sección se muestran las mejoras en el tiempo de cálculo realizados para el código de C y CUDA C. El índice *Speed Up* (SU) calcula la mejora en el tiempo de cálculo entre códigos y se obtiene mediante la Ec. (4.6), siendo t el tiempo de cálculo y p significa comparación entre precisiones. La comparación se realizó en simple y doble precisión para las dos (2) PC detalladas en la Sec. (3.1).

$$SU = \frac{t_{\,C}}{t_{\,Cuda\,C}} \qquad\qquad SU_p = \frac{t_{\,doble\,precision}}{t_{\,simple\,precision}} \tag{4.6}$$

Es de importancia resaltar que el tiempo de cálculo de la validación del problema de la Construcción de Maxwell en los 50000 pasos de tiempo con una malla de 201x201 nodos es de alrededor de 1724 segundos en el código de C, para lo que se utilizó la CPU Intel Core i7-3770 en simple precisión. Para las mismas condiciones del problema, el tiempo de cálculo de la CPU Intel Core i7-4770 es de alrededor de 1616 segundos. Por ello conocer cuánto es el SU es de gran relevancia.

Para calcular el SU de éste problema, se tomó una T_r fija y se varió el tamaño de la grilla, de manera que ésta siempre fuese cuadrada, respetando un número de nodos de potencia de 2 en los lados del cuadrado. La cantidad de *thread blocks* que se utilizó para realizar la comparación en el código de CUDA fueron de potencia de 2.

NVIDIA GeForce GTX 760

Los tamaños de grilla que se utilizaron para realizar las pruebas de tiempo de esta placa, varían entre 16x16 nodos hasta 2048x2048 nodos. La cantidad de *thread blocks* que se utilizó fueron de 16, 32, 64, 128 y 512.

Las Figuras (4.5) y (4.6) muestran el SU obtenido en simple precisión y doble precisión respectivamente. El mejor resultado en ambos casos se obtuvo para un número de *thread block* igual a 64, donde la mejora fue de 18.67 y 11.40 en simple y doble precisión respectivamente, para el mayor número de elementos de malla.

La Figura (4.7) muestra el SU_p del código de CUDA C en la GPU NVIDIA GeForce GTX 760. Para un número de *thread block* igual a 64, el tiempo de cálculo en doble precisión es 1,68 veces mayor que en simple precisión en el mayor número de elementos de malla calculado. Se reporta únicamente el resultado para esa cantidad de *thread block*, debido a que los resultados de las Figuras (4.5) y (4.6) indican que es el que tiene un mayor SU.

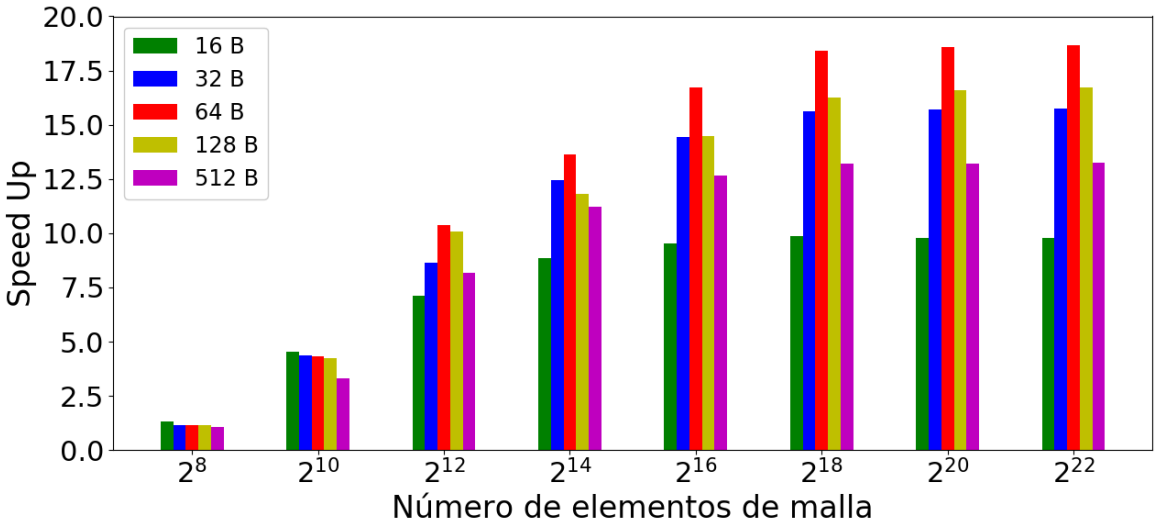


Figura 4.5: SU realizado para el problema de la Construcción de Maxwell en simple precisión con una CPU Intel Core i7-3770 y GPU NVIDIA GeForce GTX 760.

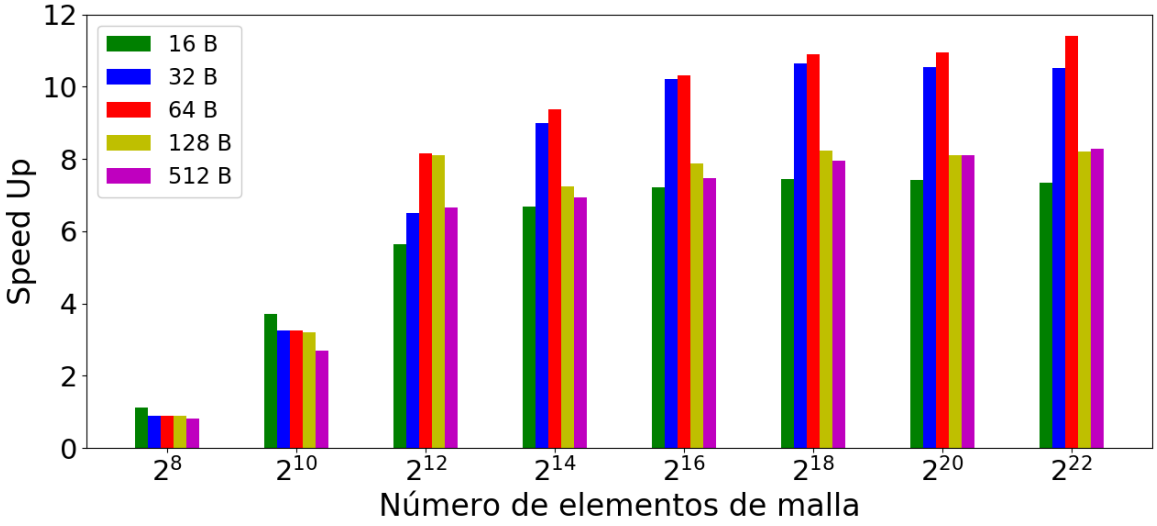


Figura 4.6: SU realizado para el problema de la Construcción de Maxwell en doble precisión con una CPU Intel Core i7-3770 y GPU NVIDIA GeForce GTX 760.

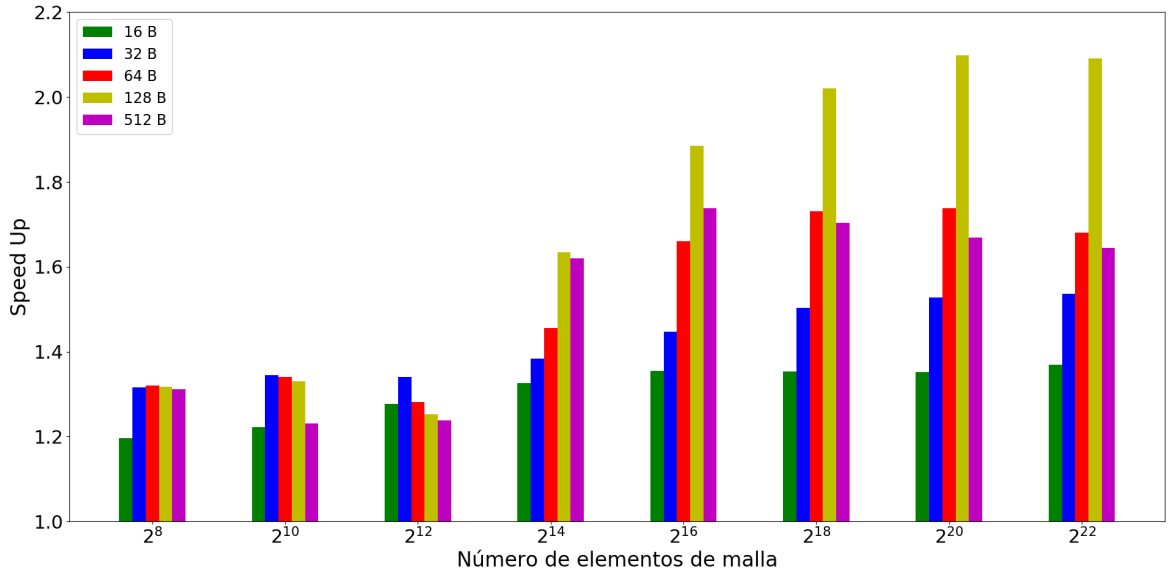


Figura 4.7: SU_p realizado para el problema de la Construcción de Maxwell en en el código de CUDA C con la GPU NVIDIA GeForce GTX 760.

NVIDIA GeForce GTX 970

Los tamaños de grilla que se utilizaron para realizar las pruebas de tiempo de esta placa, varían entre 16x16 nodos hasta 4096x4096 nodos en simple precisión y varían entre 16x16 nodos hasta 2048x2048 nodos en doble precisión . La cantidad de *thread blocks* que se utilizó fueron de 16, 32, 64, 128 y 512.

En este caso el mejor resultado obtenido de SU para simple y doble precisión fue para un número de *thread block* igual a 32. Las mejoras fueron de 23.39 y 10.96 en simple y doble precisión respectivamente, para el mayor número de elementos de malla.

En cuanto al SU_p del código de CUDA C en la GPU NVIDIA GeForce GTX 970, se obtuvo que para un número de *thread block* igual a 32, el tiempo de cálculo en doble precisión es 1,29 veces mayor que en simple precisión en el mayor número de elementos de malla calculado.

En la Tabla (4.1) se encuentran los mejores resultados obtenidos de SU_p, SU en simple y doble precisión en las PC utilizadas.

PC	*thread per block*	SU en simple precisión	SU en doble precisión	SU_p
CPU Intel Core i7-3770 GPU NVIDIA GeForce GTX 760	64	18.67	11.40	1.68
CPU Intel Core i7-4770 GPU NVIDIA GeForce GTX 970	32	23.39	10.96	1.29

Tabla 4.1: Mejores resultados de SU y SU_p obtenidos para el problema de la Construcción de Maxwell con las PC utilizadas.

A continuación se presenta la Tabla (4.2) con los tiempos de ejecución por paso temporal del algoritmo LB para el problema de la Construcción de Maxwell, para distinta cantidad de elementos de la malla. En dicha tabla se encuentran los tiempos de ejecución realizados en simple precisión para cada CPU y GPU utilizadas, en donde la GPU emplea un número de *thread block* igual a 64.

Número de elementos de malla	CPU Intel Core i7-3770	GPU NVIDIA GeForce GTX 760	CPU Intel Core i7-4770	GPU NVIDIA GeForce GTX 970
2^8	$2{,}04 \cdot 10^{-4}$	$1{,}74 \cdot 10^{-4}$	$1{,}86 \cdot 10^{-4}$	$7{,}14 \cdot 10^{-5}$
2^{10}	$8{,}12 \cdot 10^{-4}$	$1{,}87 \cdot 10^{-4}$	$7{,}47 \cdot 10^{-4}$	$8{,}04 \cdot 10^{-5}$
2^{12}	$3{,}25 \cdot 10^{-3}$	$3{,}13 \cdot 10^{-4}$	$3{,}02 \cdot 10^{-3}$	$1{,}34 \cdot 10^{-4}$
2^{14}	$1{,}33 \cdot 10^{-2}$	$9{,}77 \cdot 10^{-4}$	$1{,}22 \cdot 10^{-2}$	$5{,}35 \cdot 10^{-3}$
2^{16}	$5{,}31 \cdot 10^{-2}$	$3{,}18 \cdot 10^{-3}$	$4{,}97 \cdot 10^{-2}$	$7{,}24 \cdot 10^{-3}$
2^{18}	$2{,}15 \cdot 10^{-1}$	$1{,}16 \cdot 10^{-2}$	$2{,}00 \cdot 10^{-1}$	$1{,}54 \cdot 10^{-2}$
2^{20}	$8{,}48 \cdot 10^{-1}$	$4{,}57 \cdot 10^{-2}$	$7{,}88 \cdot 10^{-1}$	$8{,}64 \cdot 10^{-2}$
2^{22}	3,39	$1{,}82 \cdot 10^{-1}$	3,19	$2{,}36 \cdot 10^{-1}$
2^{24}	-	-	$1{,}27 \cdot 10^{1}$	$6{,}02 \cdot 10^{-1}$

Tabla 4.2: Tiempo de ejecución (segundos) por paso temporal del algoritmo de LB para el problema de la Construcción de Maxwell en simple precisión, según las CPU y GPU utilizadas. El tiempo de ejecución se encuentra para un número de elementos de malla y la cantida de *threads per block* utilizados fue de 64 en las GPU.

Se puede concluir después de haber realizado un SU y SU_p a las dos PC que se tuvo acceso para realizar el presente trabajo, de que la utilización de simple precisión en el código de Cuda C es más conveniente que doble precisión. Una de las razones es que no hay demasiada diferencia entre los valores que se pueden obtener, según los resultados obtenidos no difirieren más del 0,003 %. Otra de las razones es que debido a las mejoras obtenidas en los tiempos de cálculo en las placas NVIDIA GeForce GTX 760 y NVIDIA GeForce GTX 970 en el código de Cuda C de 18.67 y 23.39 respectivamente en simple precisión que el código de C. Las mejoras en doble precisión de las ganancias son de 11.40 y 10.96 respectivamente en las placas mencionadas. Por lo que el resultado obtenido en simple precisión difiere apenas un 0.003 % que en doble precisión, además siendo 1.68 y 1.29 veces más rapido según la GPU utilizada.

4.2. Estratificación de un fluido VdW con temperatura no uniforme (1D)

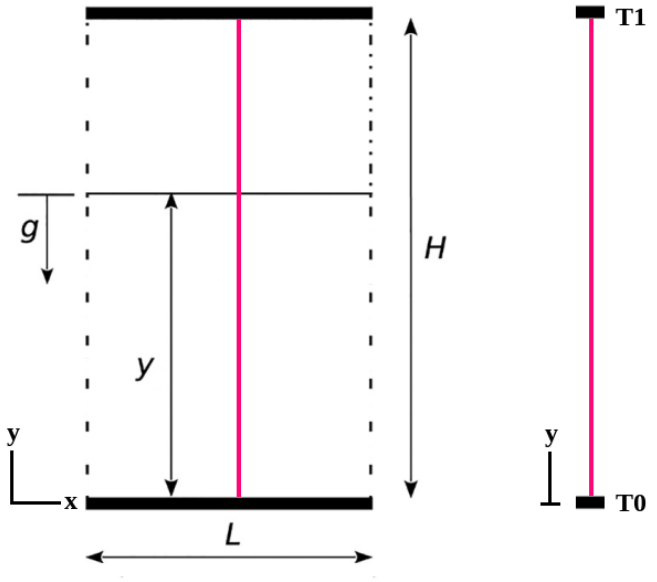


Figura 4.8: Izquierda: Cavidad bidimensional de alto (H) y ancho (L), la linea vertical indica el problema unidimensional. Derecha: Cavidad unidimensional de alto (H). Ambas cavidades bajo la acción de la gravedad (g) y temperaturas fijas en los extremos de H.

Se quiere resolver el problema de una cavidad unidimensional en presencia de un fluido cuya EOS es la de Van der Waals; con temperatura no uniforme y fuerza de gravedad no nula. Este problema fue desarrollado por Berberan-Santos et al. [35] y extendido por Fogliatto et al. [9]. En este caso se resuelven las dos ecuaciones pseudo-potencial del LBM, la ecuación hidrodinámica y la ecuación de energía.

Se toma como coordenada del problema y, teniéndose una temperatura fija T_0 en $y = 0$ y T_1 en $y = H$ como se observa en el panel derecho de la Figura (4.8) en presencia de la gravedad.

El gradiente de presión surge de realizar el balance de momento en un volumen diferencial, obteniéndose:

$$\frac{dP}{dy} = -gMC(y) \tag{4.7}$$

siendo g la gravedad, M el peso molecular, $C = \frac{1}{v}$ la fracción molar, v el volumen molar y P la presión.

Realizando la adimensionalización $P_r = \frac{P}{P_c}$, $T_r = \frac{T}{T_c}$, $c = Cv_c$ y $E_r = \frac{Mgy}{RT_c}$; siendo c el punto crítico en el cuál comienza la coexistencia de las dos fases, se pueden reemplazar en el gradiente de presión una Ecuación de estado para obtener una ecuación

adimensional con la concentración molar distribuida [9].

En particular, para la EOS de VdW se obtiene:

$$\frac{dc}{dE_r} = -\left[c + \frac{dT_r}{dE_r}\left(\frac{c}{1-\frac{c}{3}}\right)\right]\left[\frac{1}{\frac{T_r}{\left(1-\frac{c}{3}\right)^2} - \frac{9}{4}c}\right] \tag{4.8}$$

donde la posición de la interface dentro de la cavidad se puede determinar utilizando la masa inicial, siendo la misma una condición inicial para realizar la integración de la Ec. (4.8). Si se elige resolver iterativamente la Ec. (4.8) junto con la ecuación macroscópica Ec. (2.26) (ecuación del calor) se puede obtener las distribuciones de densidad y temperatura de la cavidad [9]. A partir de la densidad y temperatura recuperadas, se obtienen dos curvas adimensionales correspondientes al perfil de temperatura y al de concentraciones.

En éste problema a validar, se toma como temperatura fija $T_1 = 0{,}99\,T_c$, y se hace variar el valor de T_0. Se asigna la siguiente nomenclatura $\rho_r = \frac{\rho}{\rho_c}$ y $Y_r = \frac{y}{H}$, que en este caso coinciden con c y $\frac{E_r}{E_{r_{max}}}$ respectivamente. En las Figuras (4.9) y (4.10) se muestran las curvas adimensionales de $\rho_r - Y_r$ y $T_r - Y_r$ respectivamente.

4.2.1. Validación

La validación de este problema se hizo utilizando los parámetros $a = 0,5$ y $b = 4,0$; para un tamaño de malla de 3 x 300 nodos y $T_0 = T_r$, con $T_r = 0,6; 0,7; 0,8\, y\, 0,9$. Los parámetros de LBM son los que se detallan a continuación:

$$diag(\mathbf{\Lambda}) = \begin{bmatrix}1{,}0 & 0{,}8 & 1{,}1 & 1{,}0 & 1{,}1 & 1{,}0 & 1{,}1 & 0{,}8 & 0{,}8\end{bmatrix}$$
$$diag(\mathbf{Q}) = \begin{bmatrix}1{,}0 & 1{,}0 & 1{,}0 & 0{,}8 & 1{,}0 & 0{,}8 & 1{,}0 & 1{,}0 & 1{,}0\end{bmatrix}$$
$$\alpha_1 = 1{,}0 \qquad \alpha_2 = 1{,}0 \qquad C_v = 1{,}0$$
$$G = -1{,}0 \quad c = 1{,}0 \quad \sigma = 0{,}125 \quad a = 0{,}5 \quad b = 4{,}0$$
$$\mathbf{g} = (0{,}0 \quad -1{,}234567e^{-7} \quad 0{,}0) \qquad \rho_c = \frac{1}{12} \qquad T_c = 0{,}037037037$$

Las Figuras (4.9) y (4.10) muestran la validación del código realizado en C para simple precisión en una CPU Intel Core i7-3770 con distintos valores de T_0.

El quiebre abrupto y discontinuo de la densidad a lo largo de la columna en la Figura (4.9) muestra la coexistencia de las dos fases, para la solución analítica. Debido a que se tiene una solución continua y difusa porque el modelo toma una cierta cantidad de nodos como separación entre ambas fases, se presenta una mayor diferencia en el

ajuste de la curva obtenida con la analítica en el cambio de fase. Lo mismo sucede en la Figura (4.10). En todos los casos se simularon 750000 pasos de tiempo hasta alcanzar una solución estacionaria. El resultado obtenido para ambas curvas validades es similar en primer orden en el código de C y CUDA C.

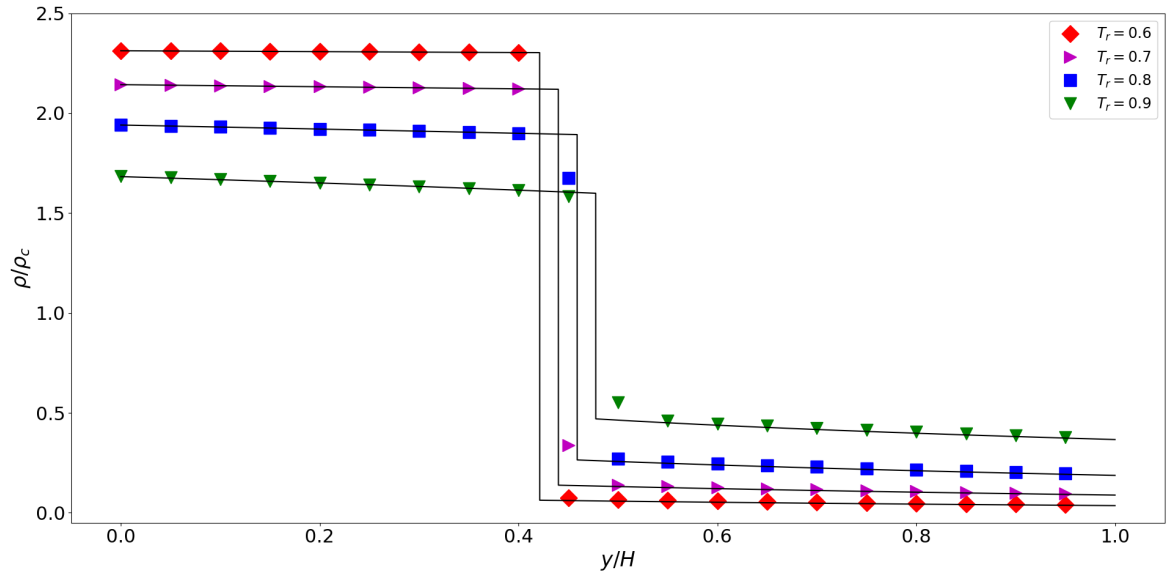


Figura 4.9: Perfil de densidad adimensional a lo largo de la cavidad, para valores de $T_0 = T_r$, siendo $T_r = 0,6\,;\ 0,7\,;\ 0,8$ *y* $0,9$, para un fluido de VdW con los parámetros $a = 0,5$ y $b = 4,0$, obtenida en simple precisión en la CPU Intel Core i7-3770 en el código desarrollado en C.

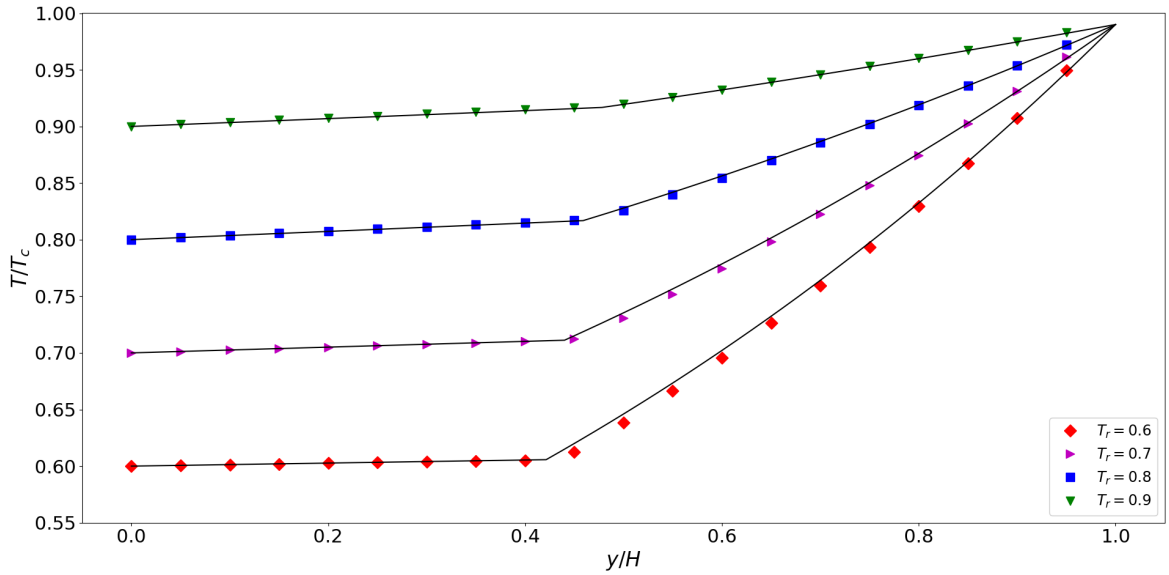


Figura 4.10: Perfil de temperatura adimensional a lo largo de la cavidad de la pared, para valores de $T_0 = T_r$, siendo $T_r = 0,6\,;\ 0,7\,;\ 0,8$ *y* $0,9$, para un fluido de VdW con los parámetros $a = 0,5$ y $b = 4,0$, obtenida en simple precisión en la CPU Intel Core i7-3770 en el código desarrollado en C.

4.2.2. Speed Up

En la presente sección se muestran las mejoras en el tiempo de cálculo realizados para el código de C y CUDA C. El índice que se utiliza es el SU y SU_p como indica la Ec. (4.6). Se tomó una T_r fija y se varió el tamaño de la grilla, de manera que ésta siempre tuviese una altura constante de 300 nodos y variando la dimensión en el eje X respetando un número de nodos de potencia de 2 (exceptuando el primer valor que es 3) . La cantidad de *thread blocks* que se utilizó para realizar la comnparación en el código de CUDA fueron de potencia de 2.

El tiempo de cálculo de la validación del presente problema en los 750000 pasos de tiempo con una malla de 3x300 nodos, fue de 776 segundos. Utilizando el código de C para la CPU Intel Core i7-3770 en simple precisión. Para las mismas condiciones del problema, el tiempo de cálculo de la CPU Intel Core i7-4770 es de alrededor de 675 segundos.

NVIDIA GeForce GTX 760

Los tamaños de grilla que se utilizaron para realizar las pruebas de tiempo de esta placa, varían entre 3x300 nodos hasta 1638400x300 nodos. La cantidad de *thread blocks* que se utilizó fueron de 16, 32, 64, 128 y 512.

Las Figuras (4.11) y (4.12) muestran el SU obtenido en simple y doble precisión respectivamente. El mejor resultado en ambos casos se obtuvo para un número de *thread block* igual a 64, donde la mejora fue de 13.26 y 7.88 en simple y doble precisión respectivamente, para el mayor número de elementos de malla.

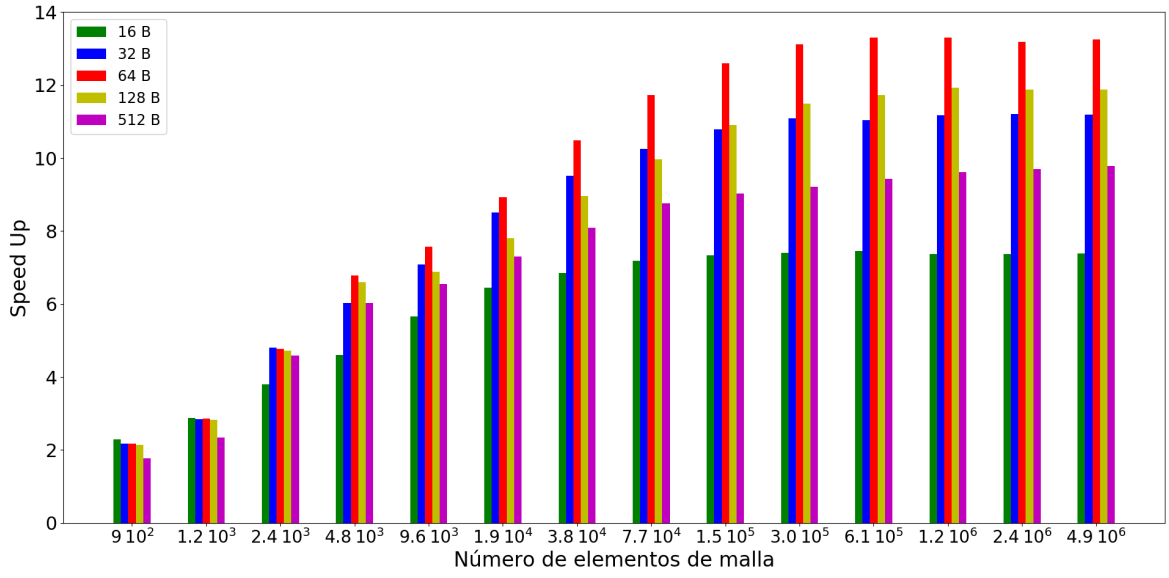


Figura 4.11: SU realizado para el problema de la Estratificación de un fluido Van dar Waals en simple precisión con una CPU Intel Core i7-3770 y GPU NVIDIA GeForce GTX 760.

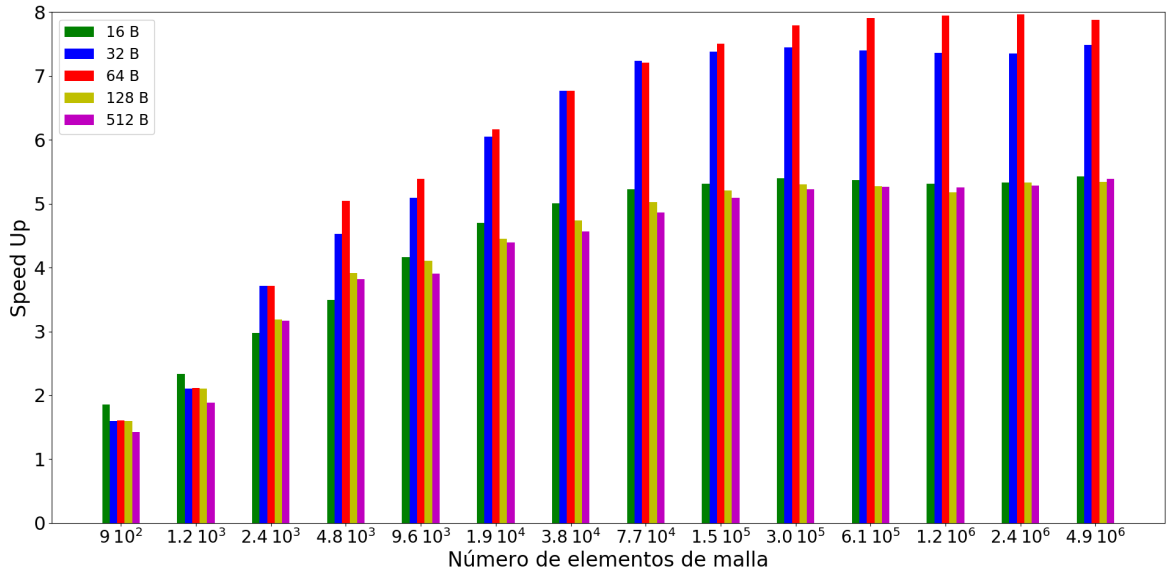


Figura 4.12: SU realizado para el problema de la Estratificación de un fluido Van dar Waals en doble precisión con una CPU Intel Core i7-3770 y GPU NVIDIA GeForce GTX 760.

La Figura (4.13) muestra el SU_p del código de CUDA C en la GPU NVIDIA GeForce GTX 760. Para un número de *thread block* igual a 64, el tiempo de cálculo en doble precisión es 1,77 veces mayor que en simple precisión en el mayor número de elementos de malla calculado. Se reporta únicamente el resultado para esa cantidad de *thread block*, debido a que los resultados anteriores indican que es el que tiene un mayor SU.

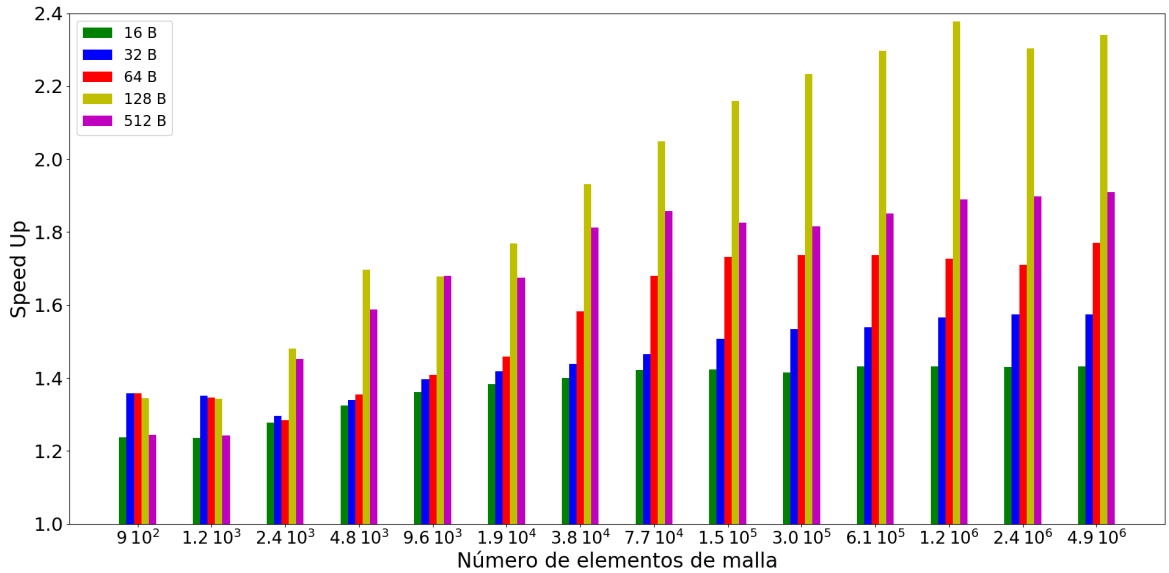


Figura 4.13: SU_p realizado para el problema de la Estratificación de un fluido Van der Waals en en el código de CUDA C con la GPU NVIDIA GeForce GTX 760.

NVIDIA GeForce GTX 970

Los tamaños de grilla que se utilizaron para realizar las pruebas de tiempo de esta placa, varían entre 3x300 nodos hasta 1638400x300 nodos. La cantidad de *thread blocks* que se utilizó fueron de 16, 32, 64, 128 y 512.

En este caso el mejor resultado obtenido de SU para simple y doble precisión fue para un número de de *thread block* igual a 32. Las mejoras fueron de 15.95 y 13.29 en simple y doble precisión respectivamente, para el mayor número de elementos de malla.

En cuanto al SU_p del código de CUDA C en la GPU NVIDIA GeForce GTX 970, se obtuvo que para un número de *thread block* igual a 32, el tiempo de cálculo en doble precisión es 1,25 veces mayor que en simple precisión en el mayor número de elementos de malla calculado.

En la Tabla (4.3) se encuentran los mejores resultados obtenidos de SU_p, SU en simple y doble precisión en las PC utilizadas.

PC	*thread per block*	SU en simple precisión	SU en doble precisión	SU_p
CPU Intel Core i7-3770 GPU NVIDIA GeForce GTX 760	64	13.26	7.88	1.77
CPU Intel Core i7-4770 GPU NVIDIA GeForce GTX 970	32	15.95	13.29	1.25

Tabla 4.3: Mejores resultados de SU y SU_p obtenidos para el problema de la Estratificación de un fluido VdW con las PC utilizadas.

En la página siguiente se presenta la Tabla (4.4) con los tiempos de ejecución por paso temporal del algoritmo LB para el problema de la Estratificación de un fluido VdW, para distinta cantidad de elementos de la malla. En dicha tabla se encuentran los tiempos de ejecución realizados en simple precisión para cada CPU y GPU utilizadas, en donde la GPU emplea un número de *thread block* igual a 64.

Número de elementos de malla	CPU Intel Core i7-3770	GPU NVIDIA GeForce GTX 760	CPU Intel Core i7-4770	GPU NVIDIA GeForce GTX 970
$9 \cdot 10^2$	$8{,}39 \cdot 10^{-4}$	$3{,}85 \cdot 10^{-4}$	$7{,}29 \cdot 10^{-4}$	$1{,}59 \cdot 10^{-4}$
$1{,}2 \cdot 10^3$	$1{,}12 \cdot 10^{-3}$	$3{,}92 \cdot 10^{-4}$	$9{,}71 \cdot 10^{-4}$	$1{,}62 \cdot 10^{-4}$
$2{,}4 \cdot 10^3$	$2{,}24 \cdot 10^{-3}$	$4{,}70 \cdot 10^{-4}$	$1{,}94 \cdot 10^{-3}$	$1{,}94 \cdot 10^{-4}$
$4{,}8 \cdot 10^3$	$4{,}48 \cdot 10^{-3}$	$6{,}60 \cdot 10^{-3}$	$4{,}05 \cdot 10^{-3}$	$1{,}83 \cdot 10^{-3}$
$9{,}6 \cdot 10^3$	$8{,}96 \cdot 10^{-3}$	$1{,}83 \cdot 10^{-3}$	$7{,}78 \cdot 10^{-3}$	$4{,}85 \cdot 10^{-3}$
$1{,}9 \cdot 10^4$	$1{,}81 \cdot 10^{-2}$	$2{,}02 \cdot 10^{-3}$	$1{,}58 \cdot 10^{-2}$	$3{,}11 \cdot 10^{-3}$
$3{,}8 \cdot 10^4$	$3{,}61 \cdot 10^{-2}$	$3{,}45 \cdot 10^{-3}$	$3{,}12 \cdot 10^{-2}$	$4{,}61 \cdot 10^{-3}$
$7{,}6 \cdot 10^4$	$7{,}30 \cdot 10^{-2}$	$6{,}22 \cdot 10^{-3}$	$6{,}49 \cdot 10^{-2}$	$2{,}48 \cdot 10^{-2}$
$1{,}5 \cdot 10^5$	$2{,}93 \cdot 10^{-2}$	$2{,}33 \cdot 10^{-3}$	$2{,}52 \cdot 10^{-2}$	$4{,}05 \cdot 10^{-2}$
$3{,}0 \cdot 10^5$	$2{,}94 \cdot 10^{-1}$	$2{,}24 \cdot 10^{-2}$	$2{,}55 \cdot 10^{-1}$	$5{,}39 \cdot 10^{-2}$
$6{,}1 \cdot 10^5$	$1{,}17 \cdot 10^{-1}$	$8{,}82 \cdot 10^{-2}$	$1{,}01 \cdot 10^{-1}$	$1{,}85 \cdot 10^{-1}$
$1{,}2 \cdot 10^6$	1,17	$8{,}82 \cdot 10^{-2}$	1,02	$1{,}41 \cdot 10^{-1}$
$2{,}4 \cdot 10^6$	1,17	$8{,}90 \cdot 10^{-2}$	1,03	$1{,}64 \cdot 10^{-1}$
$4{,}9 \cdot 10^6$	4,92	$3{,}54 \cdot 10^{-1}$	4,09	$2{,}69 \cdot 10^{-1}$

Tabla 4.4: Tiempo de ejecución (segundos) por paso temporal del algoritmo de LB para el problema de la Estratificación de un fluido VdW en simple precisión, según las CPU y GPU utilizadas. El tiempo de ejecución se encuentra para un número de elementos de malla y la cantida de *threads per block* utilizados fue de 64 en las GPU.

En este problema las mejoras de implementación del código en CUDA C con respecto a C son considerables, y se puede observar en el índice SU. En el caso de la GPU NVIDIA GeForce GTX 760 se obtuvo un SU de 13,26 y 7,88 para simple y doble precisión respectivamente. Por otro lado en la GPU NVIDIA GeForce GTX 970 se obtuvo un SU de 15,95 y 13,29 en simple y doble precisión respectivamente.

Como se mecionó anteriormente, en este problema de estratficación de fluido VdW pudo observarse que la implementación en CUDA C produce una mejora significativa en el tiempo de cálculo con respecto la la implementación equivalente en C. Para este caso, sin embargo, se observó que el SU máximo es inferior al alcanzado en el problema de la Construcción de Maxwell, es decir, con una única LBE.

4.3. Generación de burbujas sobre una superficie horizontal calefaccionada (2D)

Mediante este problema se busca reproducir la fenomenología asociada a los procesos de transferencia de calor en ebullición, demostrando la potencialidad del método.

La diferencia del problema de la Estratificación de VdW con el de la Generación de burbujas, radica en que el primero es unidimensional y el segundo es bidimensional. A su vez, la aplicación del código cambia únicamente en realizar una función que imponga como condición de contorno un área de la superficie calefaccionada.

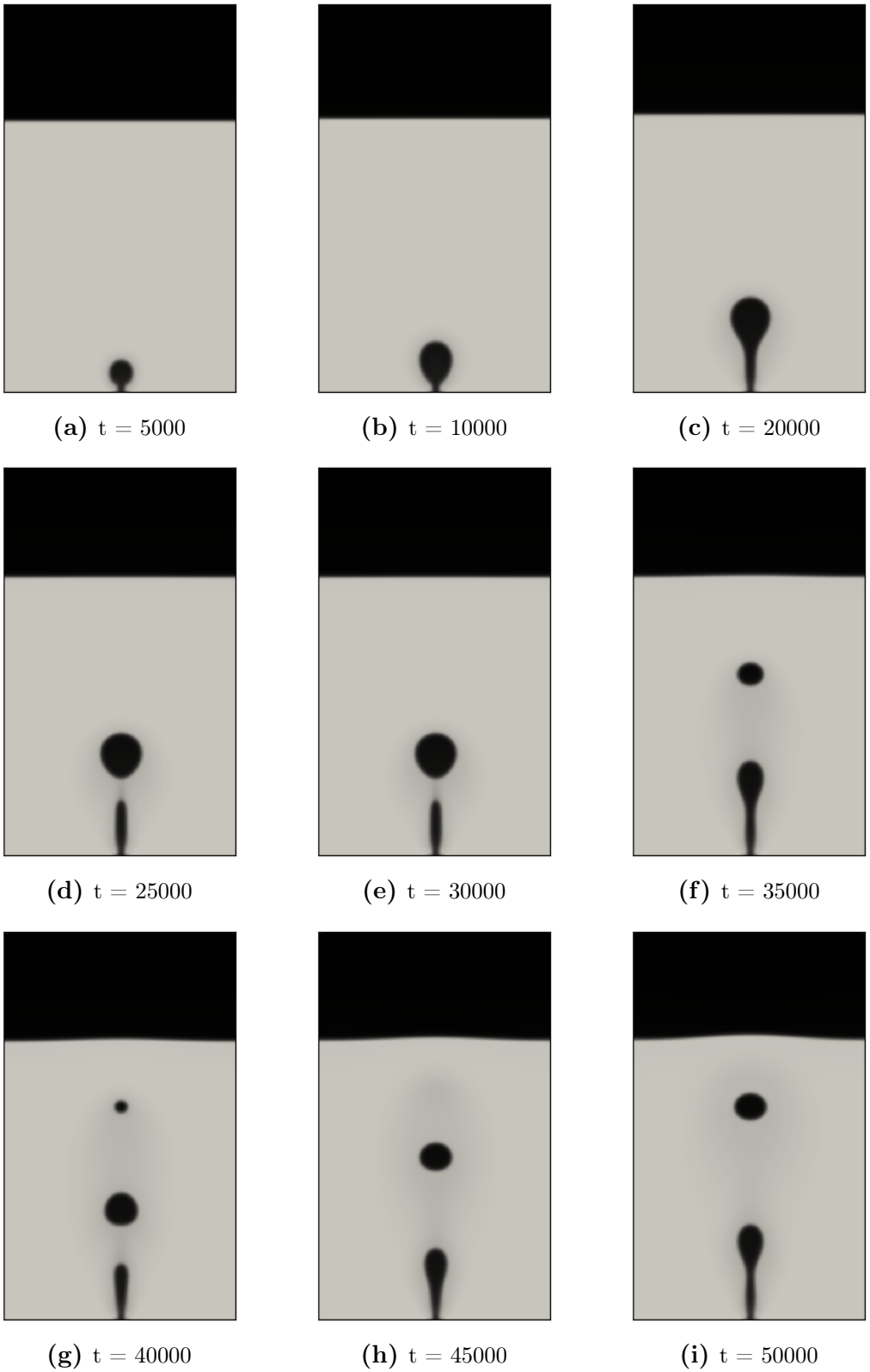
A partir de las mismas bibliotecas utilizadas anteriormente y cambiando una única función, se obtuvo como resultado el fenómeno de formación, crecimiento y desprendimiento de una burbuja. Para ello se utilizaron los siguientes parámetros de LBM:

$$
\begin{aligned}
diag(\mathbf{\Lambda}) &= \begin{bmatrix} 1{,}0 & 1{,}25 & 1{,}0 & 1{,}0 & 1{,}1 & 1{,}0 & 1{,}1 & 1{,}3 & 1{,}3 \end{bmatrix} \\
diag(\mathbf{Q}) &= \begin{bmatrix} 1{,}0 & 1{,}0 & 1{,}0 & 1{,}55 & 1{,}0 & 1{,}55 & 1{,}0 & 1{,}0 & 1{,}0 \end{bmatrix} \\
\alpha_1 &= -2{,}0 \qquad \alpha_2 = 2{,}0 \qquad C_v = 5{,}0 \\
G &= -1{,}0 \quad c = 1{,}0 \quad \sigma = 0{,}125 \quad a = 1{,}0 \quad b = 4{,}0 \\
\mathbf{g} &= (0{,}0 \quad -8{,}0^{-6} \quad 0{,}0) \qquad \rho_c = \frac{1}{12} \qquad T_c = 0{,}074074074
\end{aligned}
$$

El tamaño de grilla utilizado fue de L = 300 y H = 500 basándose en el esquema del panel izquierdo de la Figura (4.8), donde la superficie calefaccionada se encuentra en el centro del lado L y posee un tamaño de 12 elementos de malla. La temperatura de calefacción $T_{heat} = T_c$ y el resto de la superficie se encuentra a $T = 0{,}8\,T_c$ con la parte superior de la cavidad está a $T = 0{,}99\,T_c$. La separación inicial de las dos fases se encuentra en $y = 350$, donde $\rho = 0{,}1610588$ *si* $y \in [0, 350]$ y $\rho = 0{,}0199722$ y para $(350, 500]$.

El resultado que se obtuvo para 5000, 10000, 20000, 25000, 30000, 35000, 40000, 45000 y 50000 pasos de tiempo se encuentra en la Figura (4.14), realizada en la GPU NVIDIA GeForce GTX 760 para doble precisión. Adicionalmente en la Figura (4.15) se encuentra para 25000 pasos de tiempo la comparación entre simple y doble precisión realizado en la GPU NVIDIA GeForce GTX 760. También se realizó la comparación entre el resultado obtenido para 45000 pasos de tiempo entre la CPU Intel Core i7-3770 y la GPU NVIDIA GeForce GTX 760 en simple precisión en la Figura (4.16). Al analizar los resultados de la CPU y GPU con sus precisiones, y comparándolas entre todos los casos realizados, resultan similares en primera instancia.

Con ésto se demuestra que la reproducción de un fenómeno complejo como la ebullición, puede ser resuelta mediante un LBM de sencilla aplicación y a su vez con un costo computacional bajo, comparando con métodos de elementos finitos o diferencias finitas [4].

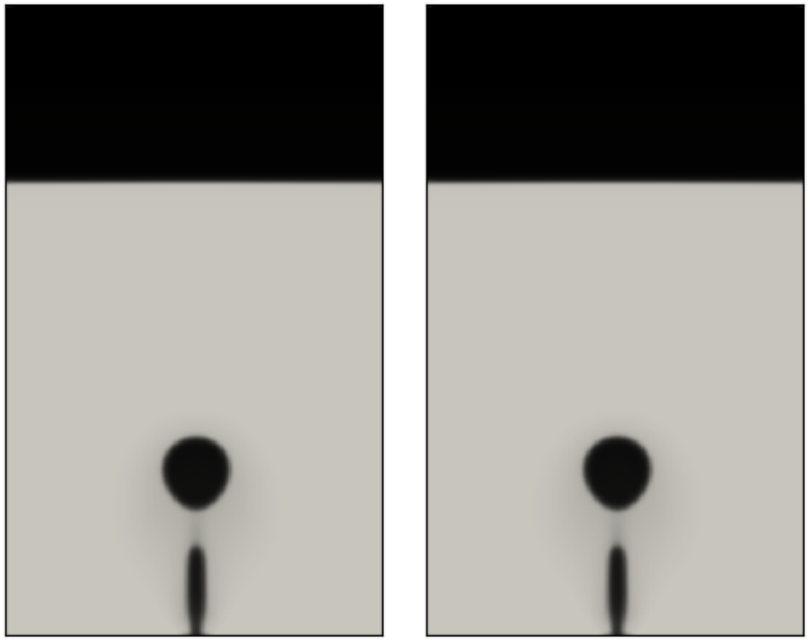


(a) t = 5000 **(b)** t = 10000 **(c)** t = 20000

(d) t = 25000 **(e)** t = 30000 **(f)** t = 35000

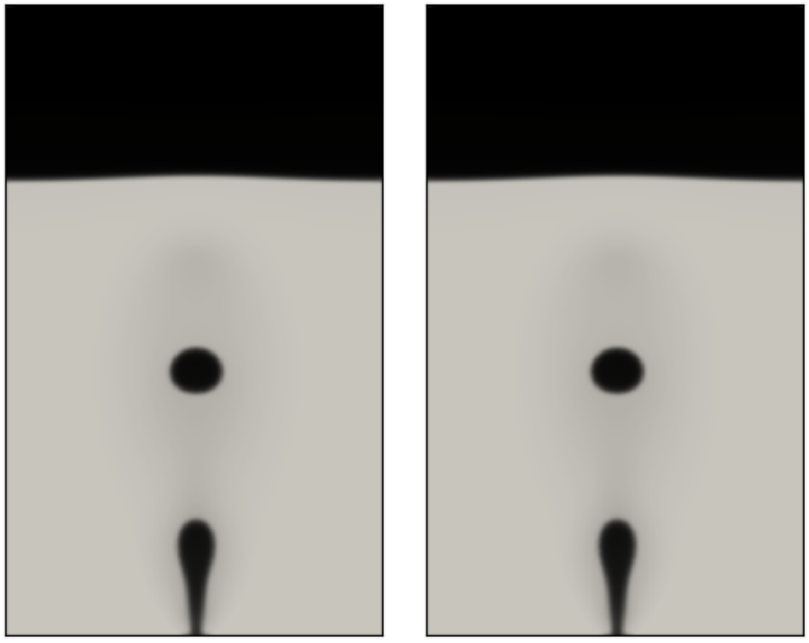
(g) t = 40000 **(h)** t = 45000 **(i)** t = 50000

Figura 4.14: Creación y desarrollo de una burbuja a partir de una superficie horizontal calefaccionada. Realizado por medio de la GPU NVIDIA GeForce GTX 760 en doble precisión.



(a) Simple precisión **(b)** Doble precisión

Figura 4.15: Creación y desarrollo de una burbuja para 25000 pasos de tiempo realizado en la GPU NVIDIA GeForce GTX 760. Izquierda: Simple precisión. Derecha: Doble precesión.



(a) Intel Core i7 - 3770 **(b)** NVIDIA GeForce GTX 760

Figura 4.16: Creación y desarrollo de una burbuja para 45000 pasos de tiempo en simple precisión. Izquierda: Realizado por medio de la CPU Intel Core i7-3770. Derecha: Realizado por medio de la GPU NVIDIA GeForce GTX 760.

4.4. Uso de PyCuda

La realización de los códigos numéricos mediante C y Cuda C tenían como primera finalidad conocer cual es la ganancia en tiempo de cálculo realizando la implementación en GPU, por otro lado desarrollar código en Cuda C, para luego compilarlo en formato ptx y ser utilizado en Python por medio del módulo PyCuda del mismo.

Debido a la falta de tiempo en el desarrollo del Proyecto Integrador no se pudo completar el desarrollo de módulos de Python que implementen con PyCuda los *kernels* compilados previemente, con el objetivo de resolver los problemas de construcción de Maxwell, estratificación de fluido VdW y generación de burbujas sobre una placa calefaccionada. Sin embargo, pudo analizarse el uso de *kernels* individuales, como el que realiza el cálculo de la densidad macroscópica (Ec. (4.9)) en todos los nodos de la grilla.

$$\rho = \sum_{\alpha} f_{\alpha} \tag{4.9}$$

Para éste análisis se inicializó a f con valores aleatorios, haciendo una repetición de la ejecución de la función 100000 veces para realizar la toma de tiempo. Se varió el tamaño de la grilla, de manera que ésta siempre fuese cuadrada, respetando un número de nodos de potencia de 2 en los lados del cuadrado. La cantidad de *thread blocks* que se utilizó para realizar la comparación en el código fue de potencia de 2. El tamaño de la grilla varió de 16x16 hasta 4096x4906 y la cantidad de *thread blocks* utilizado es de 16, 32, 64, 128 y 512.

Al realizar la implementación de un *kernel* de Cuda C en Python, se espera que el tiempo de cálculo en Python sea más lento que en Cuda C, debido a que el primer lenguaje es interpretado. Debido a lo mencionado es que se define el índice *Speed Down* (SD), donde t es el tiempo de cálculo. SD es una medida de cuánto es la demora de un código respecto a otro y se calcula de la siguiente forma:

$$SD = \frac{t_{\,PyCuda}}{t_{\,Cuda\,C}} \tag{4.10}$$

NVIDIA GeForce GTX 760

La Figura (4.17) muestra el SU para la GPU NVIDIA GeForce GTX 760, donde se observa que la mayor ganancia obtenida es de 3,2 para 64 *thread blocks*. Por otro lado, en la Figura (4.18) se muestra el SD, en donde puede verse que el menor valor alcanzado es de 1,14 para 128 y 512 *thread blocks*.

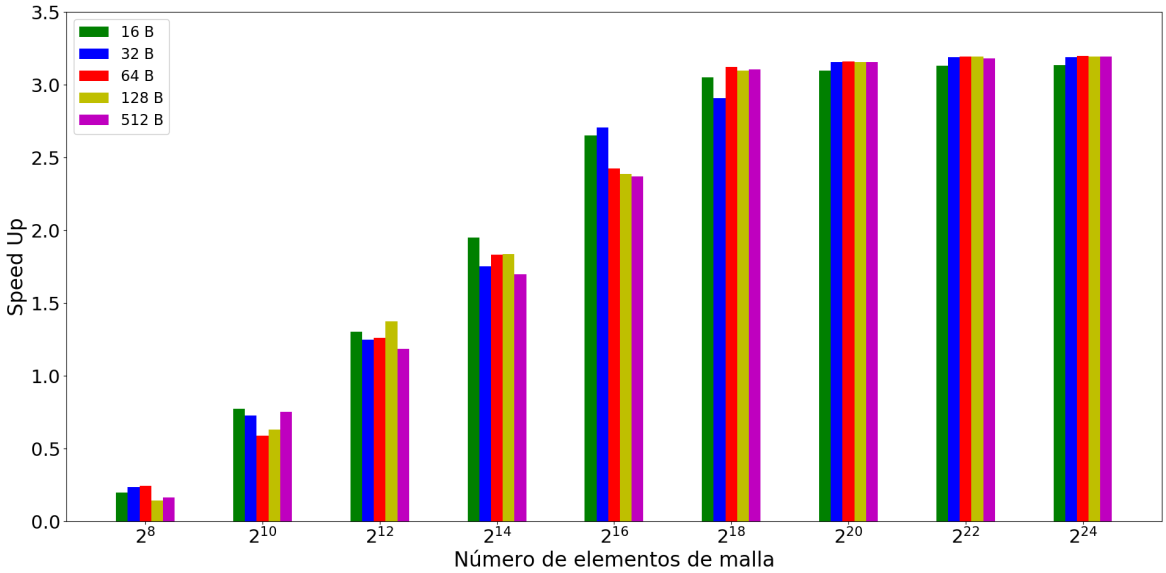


Figura 4.17: SU entre C y Cuda C para la función de obtención de la densidad con la GPU NVIDIA GeForce GTX 760 en simple precisión.

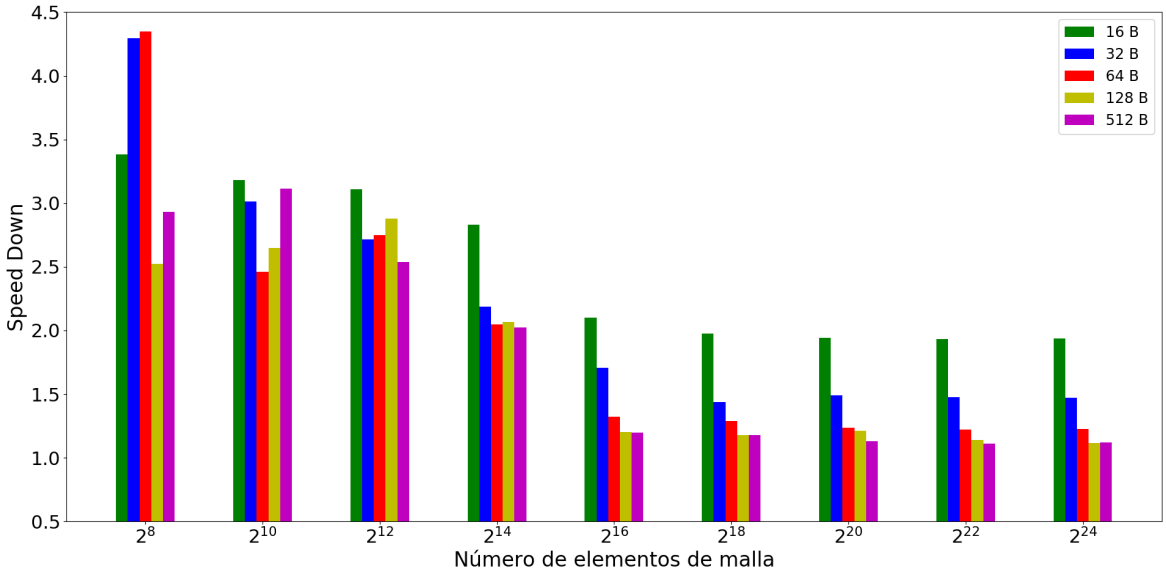


Figura 4.18: SD realizado entre PyCuda y Cuda para la función de obtención de la densidad con la GPU NVIDIA GeForce GTX 760 en simple precisión.

NVIDIA GeForce GTX 970

La Figura (4.19) muestra el SU para la GPU NVIDIA GeForce GTX 970, donde se observa que la mayor ganancia obtenida es de 4,8 para 64 *thread blocks*. Por otro lado, en la Figura (4.20) se muestra el SD, en donde puede verse que los menores valores alcanzados son 1,08 y 1,10 para 128 y 512 *thread blocks* respectivamente.

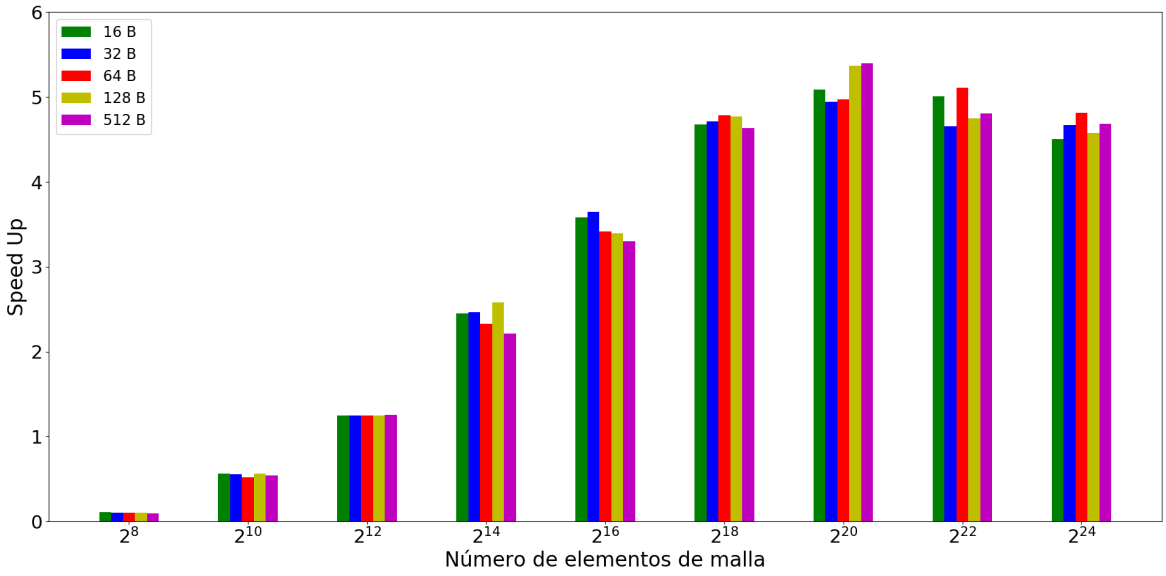


Figura 4.19: SU realizado entre C y Cuda C para la función de obtención de la densidad con la GPU NVIDIA GeForce GTX 970 en simple precisión.

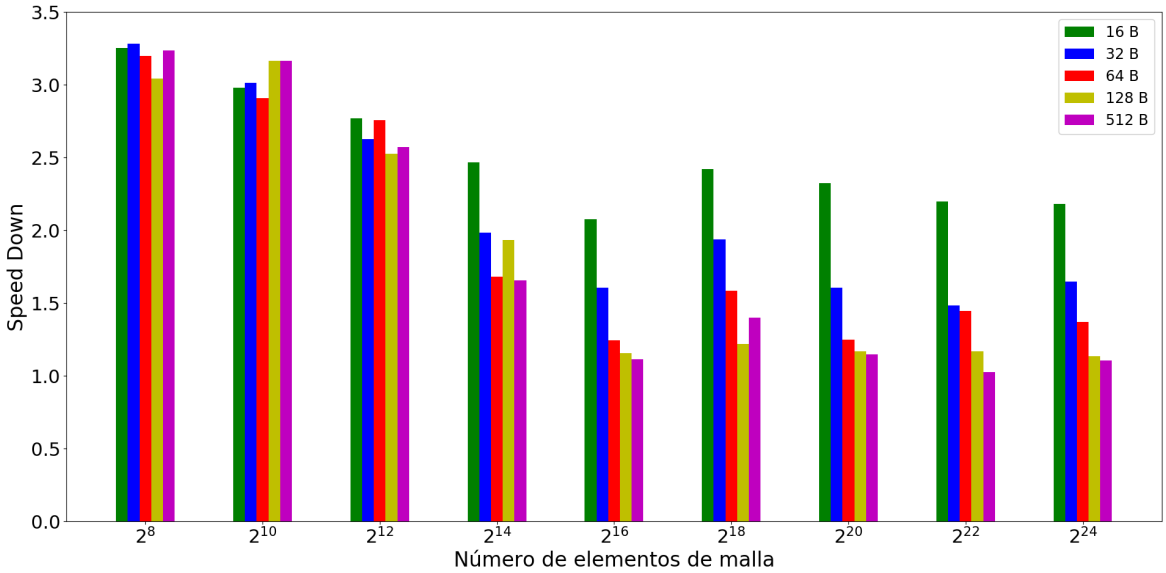


Figura 4.20: SD para la función de obtención de la densidad con la GPU NVIDIA GeForce GTX 970 en simple precisión.

Luego de haber realizado la implementación de un *kernel* de Cuda C en un *script* de Python por medio del módulo PyCuda, se observa que el tiempo de cálculo se incrementa. La variación del incremento, que se mide por el índice SD depende en gran medida de la cantidad de *thread blocks* utilizados, al igual que el tamaño de la grilla.

Debido a que se pretende utilizar con la mayor eficiencia posible los *kernels* en *Python*, debe ejecutarse con la cantidad de *thread blocks* que minimicen SD. De las pruebas realizadasse obtuvo que para la GPU NVIDIA GeForce GTX 760 con 128 y 512 *thread blocks* el valor de SD se encuentra cercano al 15 % , mientras que en la GPU NVIDIA GeForce GTX 970 ese valor esta cercano al 10 %.

Es de importancia el desarrollo de códigos numéricos para que sean eficientes en el tiempo de cálculo, y que el desarrollo del mismo no implique un gran insumo de tiempo. Por lo último mencionado realizar códigos en Python es eficiente al ser un lenguaje interpretado y no tener que reservar memoria, liberar memoria o declarar variables entre otros. Los módulos de Python tienen sus bibliotecas realizadas mayormente en C y Cuda C para que sean eficientes y debido a ello es conveniente la utilización de sus módulos, que ya poseen las funciones que se necesita, además siendo eficientes.

De esta forma, se observó que resulta posible continuar con la extensión y generalización de herramientas de cálculo eficientes utilizando una combinación adecuada de los lenguajes mencionados.

Capítulo 5

Conclusiones generales

En el presente trabajo se realizó un código numérico para resolver problemas de transferencia de calor en flujos multifásicos con cambio de fase, en donde el método utilizado es el de lattice Boltzmann de dos ecuaciones pseudo-pontencial y operador MRT desarrollado por Fogliatto et al. [10]. El modelo desarrollado resuelve problemas con discretización espacial de dominio regular y el tipo de modelo de grilla es el denominado D2Q9.

El código realizado utilizó el software CMake para preparar la compilación del mismo, el cual permite desarrollar proyectos que posean una gran cantidad de directorios de forma simple. El código se encuentra implementado en tres lenguajes de programación, siendo ellos C, Cuda C y Python. La compilación correspondiente a C se hizo en bibliotecas del tipo *shared* mientras que las bibliotecas de Cuda C eran del tipo *static*. Además los *kernels* realizados en Cuda C son compilados en formato ptx para que Python los utilice mediante su módulo llamado PyCuda.

Las instrucciones de compilación de CMake se realizan por medio de un archivo de configuración principal llamado CMakeLists.txt. Se concretaron las siguientes opciones de configuración para compilar el código:

- selección entre precisión simple o doble.
- detección automática de la arquitectura de la GPU en la PC que se compila el código.
- compilación en C o en C y Cuda C.

El código numérico fue administrado mediante el software Git que permite el control de versiones de forma fácil y eficiente. El repositorio utilizado es el de la página web GitHub y puede ser descargado en `https://github.com/efogliatto/LBCUDA_Test`. En el transcurso del proyecto se comprobó que una buena práctica para desarrollar código es mediante el uso de ramas. En este proyecto se trabajó sobre tres

ramas: *master*, *develop* y *feature*. La versión 1.0 es la que se encontraba en la rama *master* al finalizar el presente trabajo.

La validación del código fue realizado en dos PC diferentes, la primera contaba con una CPU Intel Core i7-3770 con una GPU NVIDIA GeForce GTX 760 y la segunda disponía de una CPU Intel Core i7-4770 con una GPU NVIDIA GeForce GTX 970. A su vez la validación se concretó en variables de simple precisión y doble precisión.

Los problemas físicos que se utilizaron para realizar la validación del código son los siguientes:

- Construcción de Maxwell (2D), el cual permite obtener las densidades de coexistencia de fases de un fluido a partir de la selección de una ecuación de estado para una dada condición de presión, temperatura y densidad iniciales.
- Estratificación de un fluido Van der Waals con temperatura no uniforme (1D). Este problema posee campo gravitatorio y temperaturas fijas en los extremos.
- Generación de burbujas en una placa horizontal calefaccionada (2D).

5.1. Construcción de Maxwell (2D)

Para el problema de la Construcción de Maxwell, se reprodujo el resultado que obtuvo Fogliatto et al. [9], para el cual el valor del parámetro $\sigma = 0,125$ del operador MRT es el que ajusta mejor la curva de coexistencia de fases para un fluido con la Ecuación de estado de VdW de parámetros $a = 0,5$ y $b = 4,0$. El tiempo de cálculo que lleva obtener cada una de las densidades de coexistencia de fase, fue alrededor de 1724 segundos para la CPU Intel Core i7-3770. Mientras que para la CPU Intel Core i7-4770 el tiempo de cálculo es cercano a 1616 segundos.

Para la GPU NVIDIA Geforce GTX 760 en simple precisión se obtuvo una ganancia del código realizado en CUDA C de 18.67 veces con respecto al código de **C** para un número de 64 *thread block* y la cantidad de 4194304 (2^{22}) elementos de malla. Mientras que en la GPU NVIDIA Geforce GTX 970 en las mismas condiciones se obtuvo una ganancia de 23.39 utilizando 32 *thread block* .

En doble precisión, la ganancia de la GPU NVIDIA Geforce GTX 760 con 64 *thread block* fue de 11.40 mientras que en GPU NVIDIA Geforce GTX 970 con 32 *thread block* se obtuvo 10,96. Por el comportamiento que se observó en los resultados, la GPU NVIDIA Geforce GTX 760 llegó a una ganancia máxima, mientras que la GPU NVIDIA Geforce GTX 970 posee la tendencia de aumentar su ganancia a un número de elementos de malla mayor.

Por otro lado, se compararon los resultados obtenidos en simple y doble precisión en la validación de las curvas de coexistencia, siendo el error que causa reducir la precisión de 0,003 %. Debido a que el error que introduce la precisión simple es despreciable, y puesto que el tiempo que se demora en doble precesión con respecto a simple precisión es de 1,68 y 1,29 según se utilice GPU NVIDIA Geforce GTX 760 y GPU NVIDIA Geforce GTX 970 respectivamente, se recomienda la utilización de simple precisión.

5.2. Estratificación de un fluido VdW (1D)

Para el problema unidimensional de la estratificación de un fluido VdW, se pudieron verificar los perfiles de densidad ρ_r y de temperatura T_r a lo largo de la cavidad. En donde el tiempo de cálculo que lleva realizar cada perfil es cercano a 776 segundos para la CPU Intel Core i7-3770. Mientras que para la CPU Intel Core i7-4770 el tiempo de cálculo escercano a 675 segundos.

La mayor ganancia que se obtuvo para la GPU NVIDIA Geforce GTX 760 y GPU NVIDIA Geforce GTX 970 en simple precisión fue de 13.26 y 15.95, siendo para doble precisión en 7.88 y 13.29, tomando como comparación el código de Cuda C con el código de C. En todos los casos con 64 *thread block*.

Por otro lado, para una cantidad de 32 *thread block* el código de Cuda C en simple precisión es 1.77 y 1.25 veces más rápido que en doble precisión para las GPU NVIDIA Geforce GTX 760 y GPU NVIDIA Geforce GTX 970 respectivamente.

Cabe destacar que en el problema de estratificación, que implica la resolución de dos ecuaciones, se observaron ganancias inferiores respecto a los problemas con una única ecuación.

5.3. Generación de burbujas en una superficie horizontal calefaccionada (2D)

A partir del problema de la estratificación de un fluido VdW con temperatura no uniforme, el cual es unidimensional, se pudo realizar una pequeña modificación en el código, para agregar una condición de contorno de calefacción. En esencia el código es exactamente el mismo y puede reproducir el comportamiento de generación de una burbuja en el proceso de ebullición. Por lo tanto, esto demuestra que este fenómeno complejo puede ser resuelto mediante LBM como el utilizado en este trabajo.

5.4. Eficiencia en PYTHON

En el presente trabajo se implementó mediante el módulo PYCUDA de PYTHON uno de los *kernels* del código de CUDA C, y así probar la performance del código con la utilización de dicho módulo. Se comparó el tiempo de cálculo de PYTHON para el *kernel* obtenido respecto al tiempo de cálculo del mismo en CUDA C en simple precisión. El resultado obtenido es que el incremento porcentual del tiempo de cálculo del código en PYTHON con respecto al código de CUDA C es de 15 % y 10 % para las GPU NVIDIA Geforce GTX 760 y GPU NVIDIA Geforce GTX 970 respectivamente.

5.5. Trabajo futuro

• Una de las líneas de desarrollo para este trabajo es la mejora en los *kernel* del código de CUDA C para que la ganancia en los tiempos de cálculo con respecto al del código de C aumente. Una de las formas de mejorar el rendimiento puede ser mediante la utilización de *shared memory* ó *local memory* de los *thread block*.

• El código fue realizado para que la implementación en un modelo de grilla D3Q15 pueda hacerse de manera sencilla, sin involucrar grandes modificaciones, en donde sólo se tiene que agregar en el directorio *latticeModel* este tipo de modelo.

• A partir de la implementación en el módulo PYCUDA de PYTHON de uno de los *kernels* del código de CUDA C, se puede realizar la implementación completa del código en PYTHON.

• También se puede realizar una interfaz gráfica mediante PYTHON para que el usuario pueda operar el código sin necesidad de saber utilizar la terminal. La compilación del código puede realizarse de forma tal que pueda usarse en el sistema operativo *Windows*, ya que actualmente se utiliza en *Linux*.

Apéndice A

Actividades relacionadas con la Práctica Profesional Supervisada y de Proyecto y Diseño

A.1. Práctica profesional supervisada

La práctica profesional supervisada se llevó a cabo en el Departamento de Mecánica Computacional del Centro Atómico Bariloche, durante el último año de Ingeniería Mecánica. La misma fue supervisada por el Mgter. Ezequiel Fogliatto y el Ing. Pablo Argañaras.

A.2. Proyecto y diseño

Las actividades de proyecto y diseño (PyD) realizadas para llevar a cabo el presente Proyecto Integrador de la carrera Ingeniería Mecánica fueron:

1. Identificación de la fenomenología a resolver, desarrollado en el Capítulo 2.
2. Implementación de una herramienta computacional combinando C, CUDA C y Python, explicada en el Capítulo 3.
3. Validación de la herramienta mediante la resolución de problemas con solución analítica o conocida, desarrollada en el Capítulo 4.

Presentaciones en congresos asociadas a este proyecto integrador

1. Argañaras P.E., Fogliatto E.O., Coronel T. Junio 2020. Modelo lattice Boltzmann para flujo multifásico con transferencia de calor en GPU. XXII Workshop de Investigadores en Ciencias de la Computación (WICC 2020, El Calafate, Argentina)

Bibliografía

[1] Incropera, F. P., Lavine, A. S., Bergman, T. L., DeWitt, D. P. Fundamentals of heat and mass transfer. Wiley, 2007.

[2] Schiffer, H.-W., Kober, T., Panos, E. World energy council's global energy scenarios to 2060. *Zeitschrift für Energiewirtschaft*, **42** (2), 91–102, 2018.

[3] Zhang, Y., Zhang, C., Jiang, J. Numerical simulation of fluid flow and heat transfer of supercritical fluids in fuel bundles. *Journal of nuclear science and technology*, **48** (6), 929–935, 2011.

[4] Guo, Z., Shu, C. Lattice Boltzmann method and its applications in engineering, tomo 3. World Scientific, 2013.

[5] Kelley, C. T. Iterative methods for linear and nonlinear equations, tomo 16. Siam, 1995.

[6] Krüger, T., *et al.* The lattice boltzmann method: Principles and practice. sl: Springer international publishing, 2017.

[7] Rinaldi, P. R. Modelos de autómatas celulares sobre unidades de procesamiento gráfico de alta performance. Tesis Doctoral, Universidad Nacional de Cuyo, 2011.

[8] Fogliatto, E. O., Teruel, F. E., Clausse, A. Modelado y simulación de flujo multifásico mediante el método de lattice-boltzmann. *Mecánica Computacional*, **36** (23), 1071–1077, 2018.

[9] Fogliatto, E. O., Clausse, A., Teruel, F. E. Simulation of phase separation in a van der waals fluid under gravitational force with lattice boltzmann method. *International Journal of Numerical Methods for Heat & Fluid Flow*, 2019.

[10] Fogliatto, E. O., Teruel, F. E., Clausse, A. Transferencia de calor en flujo multifásico mediante el método de lattice-boltzmann. *XXIV Congreso sobre Métodos Numéricos y sus Aplicaciones. 5 - 7 de Noviembre, 2019. Santa Fe, Argentina.*, 2019.

[11] Gunstensen, A. K., Rothman, D. H., Zaleski, S., Zanetti, G. Lattice boltzmann model of immiscible fluids. *Physical Review A*, **43** (8), 4320, 1991.

[12] Shan, X., Chen, H. Lattice boltzmann model for simulating flows with multiple phases and components. *Physical review E*, **47** (3), 1815, 1993.

[13] Swift, Y., Osborn. Lattice boltzmann simulation of nonideal fluids. *Physical review Lett*, **75**, 830, 1995.

[14] Wang, H., Yuan, X., Liang, H., Chai, Z., Shi, B. A brief review of the phase-field-based lattice boltzmann method for multiphase flows. *Capillarity*, **2** (3), 33–52, 2019.

[15] PARRILL, A. L., LIPKOWITZ, K. B. Reviews in computational chemistry, volume 31, 2019.

[16] Huang, H., Sukop, M., Lu, X. Multiphase lattice Boltzmann methods: Theory and application. John Wiley & Sons, 2015.

[17] Li, Q., Luo, K., Li, X. Lattice boltzmann modeling of multiphase flows at large density ratio with an improved pseudopotential model. *Physical Review E*, **87** (5), 053301, 2013.

[18] Succi, S., Succi, S. The lattice Boltzmann equation: for complex states of flowing matter. Oxford University Press, 2018.

[19] Márkus, A., Házi, G. Simulation of evaporation by an extension of the pseudopotential lattice boltzmann method: A quantitative analysis. *Physical Review E*, **83** (4), 046705, 2011.

[20] Zou, Q., He, X. On pressure and velocity boundary conditions for the lattice boltzmann bgk model. *Physics of fluids*, **9** (6), 1591–1598, 1997.

[21] Inamuro, T., Yoshino, M., Inoue, H., Mizuno, R., Ogino, F. A lattice boltzmann method for a binary miscible fluid. *Journal of Computational Physics*, **179** (1), 201–215, 2002.

[22] AMD. Especificaciones de procesadores, 2020. URL `https://www.amd.com/es/products/specifications/processors/1896,2466,11776,1736,2481`, [Web; accedido el 06-05-2020].

[23] Intel. Intel® core™ processor family, 2020. URL `URL{https://www.intel.la/content/www/xl/es/products/processors/core.html}`, [Web; accedido el 06-05-2020].

[24] Rajagopal, R. Introduction to Microsoft Windows NT cluster server: Programming and administration. CRC Press, 1999.

[25] Tölke, J. Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia. *Computing and Visualization in Science*, **13** (1), 29, 2010.

[26] Intel. Procesador intel® core™ i7-3770, 2020. URL `https://ark.intel.com/content/www/us/en/ark/products/65719/intel-core-i7-3770-processor-8m-cache-up-to-3-90-ghz.html`, [Web; accedido el 12-06-2020].

[27] Intel. Procesador intel® core™ i7-4770, 2020. URL `https://ark.intel.com/content/www/es/es/ark/products/75122/intel-core-i7-4770-processor-8m-cache-up-to-3-90-ghz.html`, [Web; accedido el 12-06-2020].

[28] NVIDIA. Geforce gtx 760, 2020. URL `https://www.nvidia.es/gtx-700-graphics-cards/gtx-760/`, [Web; accedido el 12-06-2020].

[29] NVIDIA. Geforce gtx 970, 2020. URL `https://www.nvidia.es/gtx-900-graphics-cards/gtx-970/`, [Web; accedido el 12-06-2020].

[30] Nobile, M. S., Cazzaniga, P., Besozzi, D., Pescini, D., Mauri, G. cutauleaping: A gpu-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems. *PLoS One*, **9** (3), 2014.

[31] Zone, N. D. Cuda toolkit documentation. *NVIDIA,[Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.[Acedido en Abril 2020]*.

[32] Represa Pérez, C., Cámara Nebreda, J. M., Sánchez Ortega, P. L., *et al.* Introducción a la programación en cuda: v. 3.1, 2016.

[33] Kitware. Build with cmake. build with confidence., 2020. URL `https://www.kitware.com/build-with-cmake-build-with-confidence/`, [Web; accedido el 15-06-2020].

[34] Git. User manual., 2020. URL `https://git-scm.com/docs/user-manual`, [Web; accedido el 15-06-2020].

[35] Berberan-Santos, M. N., Bodunov, E. N., Pogliani, L. Liquid–vapor equilibrium in a gravitational field. *American Journal of Physics*, **70** (4), 438–443, 2002.

Índice de figuras

Índice de tablas

Agradecimientos

A lo largo de todo el tiempo que transité el camino de este sueño, ahora cumplido, se me presentaron personas, momentos y lugares. El conjunto de cada uno de esos instantes me marcó y hacen que hoy sea quien soy.

Primero quiero agradecer a Ezequiel, mi director, que me guió, motivó, tuvo paciencia, y creyó en mí para terminar este PI. En segundo lugar agradezco a Pablo, mi codirector, por haberme acompañado en el proceso. Sin ellos este trabajo no hubiera sido posible, y más aún por la forma de trabajo que tuvimos que adoptar estos últimos meses.

Agradezco a mi heroína, mi mamá Norma, por haberme brindado todo el amor, cariño, aliento, fuerzas, herramientas y el ejemplo, porque hoy soy quien soy gracias a ella. A mi papá Carlos que siempre me motivó a seguir adelante e ir detrás de mis sueños. A mis hermanitos Sonia y Leandro, ya que sin sus consejos, charlas, llamadas y peleas, la vida sería muy aburrida, gracias porque sin importar la distancia me hacen sentir su calidez y apoyo incondicional para continuar. También agradezco a mis padrinos, tíos y primos por darme ese empujoncito que necité cada vez que volvia al hogar.

A mis abuelos Kiki, Chichi, y a . que son mis tres puntitos en el cielo, que me acompañan y siguen marcando en la vida.

Gracias al IB por permitirme estudiar en el lugar que siempre soñe de chico, a los profesores que a lo largo de los tres años que estuve me marcaron en la forma de aprender y de disfrutar Bariloche. Especialmente agradezco a Beto, Marcos, Tobias, Franco, Enzo, Fede, Horacio y Nancy. Después de muchas vivencias compartida a lo largo de los tres años en el IB les agradezco a Tomasito, Tincho, Santi, Lucas, Lucca, Eze y Belén, ya que los pabellones y el otoño lluviso de Brc se hizo más hogar gracias a ustedes.

Muchas gracias Norbert por el apoyo y haber estado siempre, ya sea con mates, charlas o salidas, a partir de esos primeros viajes hasta ahora. También te quiero agradecer Chipo por tus reflexiones, senderos y comidas compartidas. A Juli, Maru, Cristian y Fer que me brindaron todo su cariño y fuerzas para que pueda seguir semana a semana. A Mariano, Ernesto, Roberto, Santiago y a toda la PM550 por darme otras herramientas para pensar y progresar. Fueron mi contención en Brc.

Un agradecimiento muy particular para Tito, Iván, Adrián, Ramiro y Gabriel que siempre me esperaban con los brazos abiertos, y me hacían sentir que todavía estabamos con el overol puesto. También te agradezco a vos Tumbao que desde que tengo memoria me acompañas amigo. A mis profes del IT Blasco, Dode y Ruiz que me incentivaron a ir más allá de la conformidad. A María, Nahuel, Mapi, Juli, Ferchu y toda la comunidad de LC. A Gonzalo y Juan Cruz que me permitieron ver otro lado de la vida en éste último semestre.

A Flor y Elisa por transmitirme esa felicidad, energía y forma que poseen de ver la vida. A Pablo y Julito que me hicieron llevadero los días en los anfiteatros de la Quinta, a Matías, Flavio, Joaquín, Sabri, Leo, Blas, Franco por compartir esa etapa.

Muchas gracias a todos aquellos que de una forma u otra aparecieron en mi camino, porque durante el transcurso de este sueño, aprendí a ver y disfrutar lo maravillosa que es la vida.